# beem Documentation

***Release 0.19.23***

**Holger Nahrstaedt**

**May 09, 2018**

# Contents

Steem is a blockchain-based rewards platform for publishers to monetize content and grow community.

It is based on *Graphene* (tm), a blockchain technology stack (i.e. software) that allows for fast transactions and ascalable blockchain solution. In case of Steem, it comes with decentralized publishing of content.

The Steem library has been designed to allow developers to easily access its routines and make use of the network without dealing with all the related blockchain technology and cryptography. This library can be used to do anything that is allowed according to the Steem blockchain protocol.

CHAPTER 1

# About this Library

The purpose of *beem* is to simplify development of products and services that use the Steem blockchain. It comes with

- it's own (bip32-encrypted) wallet
- RPC interface for the Blockchain backend
- JSON-based blockchain objects (accounts, blocks, prices, markets, etc)
- a simple to use yet powerful API
- transaction construction and signing
- push notification API
- *and more*

# CHAPTER 2

# Quickstart

**Note:**

**All methods that construct and sign a transaction can be given** the `account=` parameter to identify the user that is going to affected by this transaction, e.g.:

- the source account in a transfer
- the accout that buys/sells an asset in the exchange
- the account whos collateral will be modified

**Important**, If no `account` is given, then the `default_account` according to the settings in `config` is used instead.

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("wallet-passphrase")
account = Account("test", steem_instance=steem)
account.transfer("<to>", "<amount>", "<asset>", "<memo>")
```

```
from beem.blockchain import Blockchain
blockchain = Blockchain()
for op in Blockchain.ops():
    print(op)
```

```
from beem.block import Block
print(Block(1))
```

```
from beem.account import Account
account = Account("test")
print(account.balances)
for h in account.history():
    print(h)
```

```
from beem.steem import Steem
stm = Steem()
stm.wallet.wipe(True)
stm.wallet.create("wallet-passphrase")
stm.wallet.unlock("wallet-passphrase")
stm.wallet.addPrivateKey("512345678")
stm.wallet.lock()
```

```
from beem.market import Market
market = Market("SBD:STEEM")
print(market.ticker())
market.steem.wallet.unlock("wallet-passphrase")
print(market.sell(300, 100)  # sell 100 STEEM for 300 STEEM/SBD
```

CHAPTER 3

# General

## 3.1 Installation

Warning: install beem will install pycrytodome which is not compatible to pycryto which is need for python-steem. At the moment, either beem or steem can be install at one maschine!

For Debian and Ubuntu, please ensure that the following packages are installed:

```
sudo apt-get install build-essential libssl-dev python-dev
```

For Fedora and RHEL-derivatives, please ensure that the following packages are installed:

```
sudo yum install gcc openssl-devel python-devel
```

For OSX, please do the following:

```
brew install openssl
export CFLAGS="-I$(brew --prefix openssl)/include $CFLAGS"
export LDFLAGS="-L$(brew --prefix openssl)/lib $LDFLAGS"
```

For Termux on Android, please install the following packages:

```
pkg install clang openssl-dev python-dev
```

Install beem by pip:

```
pip install -U beem
```

You can install beem from this repository if you want the latest but possibly non-compiling version:

```
git clone https://github.com/holgern/beem.git
cd beem
python setup.py build

python setup.py install --user
```

Run tests after install:

```
pytest
```

### 3.1.1 Manual installation:

```
$ git clone https://github.com/holgern/beem/
$ cd beem
$ python setup.py build
$ python setup.py install --user
```

### 3.1.2 Upgrade

```
$ pip install --user --upgrade
```

## 3.2 Quickstart

## 3.3 Tutorials

### 3.3.1 Bundle Many Operations

With Steem, you can bundle multiple operations into a single transactions. This can be used to do a multi-send (one sender, multiple receivers), but it also allows to use any other kind of operation. The advantage here is that the user can be sure that the operations are executed in the same order as they are added to the transaction.

```
from pprint import pprint
from beem import Steem
from beem.account import Account
from beem.comment import Comment
from beem.instance import set_shared_steem_instance

# Only for testing not a real working key
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

# set nobroadcast always to True, when testing
testnet = Steem(
    nobroadcast=True,
    bundle=True,
    keys=[wif],
)
# Set testnet as shared instance
set_shared_steem_instance(testnet)

# Account and Comment will use now testnet
account = Account("test")

# Post
c = Comment("@gtg/witness-gtg-log")
```

(continues on next page)

(continued from previous page)

```
account.transfer("test1", 1, "STEEM")
account.transfer("test2", 1, "STEEM")
account.transfer("test3", 1, "SBD")
# Upvote post with 25%
c.upvote(25, voter=account)

pprint(testnet.broadcast())
```

### 3.3.2 Simple Sell Script

```
from beem import Steem
from beem.market import Market
from beem.price import Price
from beem.amount import Amount

# Only for testing not a real working key
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

#
# Instanciate Steem (pick network via API node)
#
steem = Steem(
    nobroadcast=True,    # <<--- set this to False when you want to fire!
    keys=[wif]           # <<--- use your real keys, when going live!
)

#
# This defines the market we are looking at.
# The first asset in the first argument is the *quote*
# Sell and buy calls always refer to the *quote*
#
market = Market("SBD:STEEM",
    steem_instance=steem
)

#
# Sell an asset for a price with amount (quote)
#
print(market.sell(
    Price(100.0, "STEEM/SBD"),
    Amount("0.01 SBD")
))
```

### 3.3.3 Sell at a timely rate

```
import threading
from beem import Steem
from beem.market import Market
from beem.price import Price
from beem.amount import Amount

# Only for testing not a real working key
```

(continues on next page)

(continued from previous page)

```
wif = "5KQwrPbwdL6PhXujxW37FSSQZ1JiwsST4cqQzDeyXtP79zkvFD3"

def sell():
    """ Sell an asset for a price with amount (quote)
    """
    print(market.sell(
        Price(100.0, "SBD/STEEM"),
        Amount("0.01 STEEM")
    ))

    threading.Timer(60, sell).start()


if __name__ == "__main__":
    #
    # Instanciate Steem (pick network via API node)
    #
    steem = Steem(
        nobroadcast=True,    # <<--- set this to False when you want to fire!
        keys=[wif]           # <<--- use your real keys, when going live!
    )

    #
    # This defines the market we are looking at.
    # The first asset in the first argument is the *quote*
    # Sell and buy calls always refer to the *quote*
    #
    market = Market("STEEM:SBD",
        steem_instance=steem
    )

    sell()
```

## 3.4 beempy CLI

*beempy* is a convenient CLI utility that enables you to manage your wallet, transfer funds, check balances and more.

### 3.4.1 Using the Wallet

*beempy* lets you leverage your BIP38 encrypted wallet to perform various actions on your accounts.

The first time you use *beempy*, you will be prompted to enter a password. This password will be used to encrypt the *beempy* wallet, which contains your private keys.

You can change the password via *changewalletpassphrase* command.

```
beempy changewalletpassphrase
```

From this point on, every time an action requires your private keys, you will be prompted ot enter this password (from CLI as well as while using *steem* library).

To bypass password entry, you can set an environmnet variable `UNLOCK`.

```
UNLOCK=mysecretpassword beempy transfer <recipient_name> 100 STEEM
```

### 3.4.2 Common Commands

First, you may like to import your Steem account:

```
beempy importaccount
```

You can also import individual private keys:

```
beempy addkey <private_key>
```

Listing accounts:

```
beempy listaccounts
```

Show balances:

```
beempy balance account_name1 account_name2
```

Sending funds:

```
beempy transfer --account <account_name> <recipient_name> 100 STEEM memo
```

Upvoting a post:

```
beempy upvote --account <account_name> https://steemit.com/funny/@mynameisbrian/the-
↪content-stand-a-comic
```

### 3.4.3 Setting Defaults

For a more convenient use of `beempy` as well as the `beem` library, you can set some defaults. This is especially useful if you have a single Steem account.

```
beempy set default_account test
beempy set default_vote_weight 100

beempy config
 +---------------------+--------+
 | Key                 | Value  |
 +---------------------+--------+
 | default_account     | test   |
 | default_vote_weight | 100    |
 +---------------------+--------+
```

If you've set up your *default_account*, you can now send funds by omitting this field:

```
beempy transfer <recipient_name> 100 STEEM memo
```

### 3.4.4 Help

You can see all available commands with `beempy --help`

```
 ~ % beempy --help
Usage: cli.py [OPTIONS] COMMAND1 [ARGS]... [COMMAND2 [ARGS]...]...

Options:
  -n, --node TEXT         URL for public Steem API (e.g.
                          https://api.steemit.com)
  -o, --offline           Prevent connecting to network
  -d, --no-broadcast      Do not broadcast
  -p, --no-wallet         Do not load the wallet
  -x, --unsigned          Nothing will be signed
  -e, --expires INTEGER   Delay in seconds until transactions are supposed to
                          expire(defaults to 60)
  -v, --verbose INTEGER   Verbosity
  --version               Show the version and exit.
  --help                  Show this message and exit.

Commands:
  addkey                   Add key to wallet When no [OPTION] is given,...
  allow                    Allow an account/key to interact with your...
  approvewitness           Approve a witnesses
  balance                  Shows balance
  broadcast                broadcast a signed transaction
  buy                      Buy STEEM or SBD from the internal market...
  cancel                   Cancel order in the internal market
  changewalletpassphrase   Change wallet password
  claimreward              Claim reward balances By default, this will...
  config                   Shows local configuration
  convert                  Convert STEEMDollars to Steem (takes a week...
  createwallet             Create new wallet with password
  currentnode              Returns the current node
  delkey                   Delete key from the wallet PUB is the public...
  delprofile               Delete a variable in an account's profile
  disallow                 Remove allowance an account/key to interact...
  disapprovewitness        Disapprove a witnesses
  downvote                 Downvote a post/comment POST is...
  follow                   Follow another account
  follower                 Get information about followers
  following                Get information about following
  importaccount            Import an account using a passphrase
  info                     Show basic blockchain info General...
  interest                 Get information about interest payment
  listaccounts             Show stored accounts
  listkeys                 Show stored keys
  mute                     Mute another account
  muter                    Get information about muter
  muting                   Get information about muting
  newaccount               Create a new account
  nextnode                 Uses the next node in list
  openorders               Show open orders
  orderbook                Obtain orderbook of the internal market
  parsewif                 Parse a WIF private key without importing
  permissions              Show permissions of an account
  pingnode                 Returns the answer time in milliseconds
  power                    Shows vote power and bandwidth
  powerdown                Power down (start withdrawing VESTS from...
  powerdownroute           Setup a powerdown route
  powerup                  Power up (vest STEEM as STEEM POWER)
```

(continues on next page)

(continued from previous page)

```
  resteem                 Resteem an existing post
  sell                    Sell STEEM or SBD from the internal market...
  set                     Set default_account, default_vote_weight or...
  setprofile              Set a variable in an account's profile
  sign                    Sign a provided transaction with available...
  transfer                Transfer SBD/STEEM
  unfollow                Unfollow/Unmute another account
  updatememokey           Update an account's memo key
  upvote                  Upvote a post/comment POST is...
  votes                   List outgoing/incoming account votes
  walletinfo              Show info about wallet
  witnesscreate           Create a witness
  witnesses               List witnesses
  witnessupdate           Change witness properties
```

## 3.5 Configuration

The pysteem library comes with its own local configuration database that stores information like

- API node URL
- default account name
- the encrypted master password

and potentially more.

You can access those variables like a regular dictionary by using

```
from beem import Steem
steem = Steem()
print(steem.config.items())
```

Keys can be added and changed like they are for regular dictionaries.

If you don't want to load the `steem.Steem` class, you can load the configuration directly by using:

```
from beem.storage import configStorage as config
```

### 3.5.1 API

**class** beem.storage.**Configuration**

This is the configuration storage that stores key/value pairs in the *config* table of the SQLite3 database.

**checkBackup**()

Backup the SQL database every 7 days

**create_table**()

Create the new table in the SQLite database

**delete**(*key*)

Delete a key from the configuration store

**exists_table**()

Check if the database table exists

**get** (*key*, *default=None*)
Return the key if exists or a default value

**nodes = ['wss://steemd.privex.io', 'wss://steemd.pevo.science', 'wss://rpc.buildteam.i**
Default configuration

## 3.6 Contributing to beem

We welcome your contributions to our project.

### 3.6.1 Repository

The repository of beem is currently located at:

https://github.com/holgern/beem

### 3.6.2 Flow

This project makes heavy use of git flow. If you are not familiar with it, then the most important thing for your to understand is that:

pull requests need to be made against the develop branch

### 3.6.3 How to Contribute

0. Familiarize yourself with *contributing on github <https://guides.github.com/activities/contributing-to-open-source/>*
1. Fork or branch from the master.
2. Create commits following the commit style
3. Start a pull request to the master branch
4. Wait for a @holger80 or another member to review

### 3.6.4 Issues

Feel free to submit issues and enhancement requests.

### 3.6.5 Contributing

Please refer to each project's style guidelines and guidelines for submitting patches and additions. In general, we follow the "fork-and-pull" Git workflow.

1. **Fork** the repo on GitHub
2. **Clone** the project to your own machine
3. **Commit** changes to your own branch
4. **Push** your work back up to your fork
5. Submit a **Pull request** so that we can review your changes

NOTE: Be sure to merge the latest from “upstream” before making a pull request!

### 3.6.6 Copyright and Licensing

This library is open sources under the MIT license. We require your to release your code under that license as well.

## 3.7 Support and Questions

We have currently not setup a distinct channel for development around pysteemi. However, many of the contributors are frequently reading through these channels:

CHAPTER 4

# Packages

## 4.1 beem

### 4.1.1 beem package

#### Submodules

#### beem.account module

**class** beem.account.**Account**(*account*, *full=True*, *lazy=False*, *steem_instance=None*)

Bases: beem.blockchainobject.BlockchainObject

This class allows to easily access Account data

**Parameters**

- **account_name** (*str*) – Name of the account
- **steem_instance** (beem.steem.Steem) – Steem instance
- **lazy** (*bool*) – Use lazy loading
- **full** (*bool*) – Obtain all account data including orders, positions, etc.

**Returns** Account data

**Return type** dictionary

**Raises** ***beem.exceptions.AccountDoesNotExistsException*** – if account does not exist

Instances of this class are dictionaries that come with additional methods (see below) that allow dealing with an account and it's corresponding functions.

```
from beem.account import Account
account = Account("test")
print(account)
print(account.balances)
```

**Note:** This class comes with its own caching function to reduce the load on the API server. Instances of this class can be refreshed with `Account.refresh()`.

**allow** (*foreign*, *weight=None*, *permission='posting'*, *account=None*, *threshold=None*, ***kwargs*)
Give additional access to an account by some other public key or account.

**Parameters**

- **foreign** (*str*) – The foreign account that will obtain access
- **weight** (*int*) – (optional) The weight to use. If not define, the threshold will be used. If the weight is smaller than the threshold, additional signatures will be required. (defaults to threshold)
- **permission** (*str*) – (optional) The actual permission to modify (defaults to `active`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)
- **threshold** (*int*) – The threshold that needs to be reached by signatures to be able to interact

**approvewitness** (*witness*, *account=None*, *approve=True*, ***kwargs*)
Approve a witness

**Parameters**

- **witnesses** (*list*) – list of Witness name or id
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

**available_balances**
List balances of an account. This call returns instances of `steem.amount.Amount`.

**balances**

**cancel_transfer_from_savings** (*request_id*, *account=None*)
Cancel a withdrawal from 'savings' account. :param str request_id: Identifier for tracking or cancelling the withdrawal :param str account: (optional) the source account for the transfer if not `default_account`

**claim_reward_balance** (*reward_steem='0 STEEM'*, *reward_sbd='0 SBD'*, *reward_vests='0 VESTS'*, *account=None*)
Claim reward balances. By default, this will claim `all` outstanding balances. To bypass this behaviour, set desired claim amount by setting any of *reward_steem*, *reward_sbd* or *reward_vests*. Args:

reward_steem (string): Amount of STEEM you would like to claim. reward_sbd (string): Amount of SBD you would like to claim. reward_vests (string): Amount of VESTS you would like to claim. account (string): The source account for the claim if not `default_account` is used.

**convert** (*amount*, *account=None*, *request_id=None*)
Convert SteemDollars to Steem (takes one week to settle) :param float amount: number of VESTS to withdraw :param str account: (optional) the source account for the transfer if not `default_account` :param str request_id: (optional) identifier for tracking the conversion'

**curation_stats**()
Returns the curation reward of the last 24h and 7d and the average of the last 7 days

**delegate_vesting_shares**(*to_account*, *vesting_shares*, *account=None*)
Delegate SP to another account. Args:

to_account (string): Account we are delegating shares to (delegatee). vesting_shares (string): Amount of VESTS to delegate eg. *10000 VESTS*. account (string): The source account (delegator). If not specified, `default_account` is used.

**disallow**(*foreign*, *permission='posting'*, *account=None*, *threshold=None*, ***kwargs*)
Remove additional access to an account by some other public key or account.

**Parameters**

- **foreign** (*str*) – The foreign account that will obtain access
- **permission** (*str*) – (optional) The actual permission to modify (defaults to `active`)
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)
- **threshold** (*int*) – The threshold that needs to be reached by signatures to be able to interact

**disapprovewitness**(*witness*, *account=None*, ***kwargs*)
Disapprove a witness

**Parameters**

- **witnesses** (*list*) – list of Witness name or id
- **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

**ensure_full**()

**follow**(*other, what=['blog'], account=None*)
Follow/Unfollow/Mute/Unmute another account's blog :param str other: Follow this account :param list what: List of states to follow.

`['blog']` means to follow `other`, `[]` means to unfollow/unmute `other`, `['ignore']` means to ignore `other`, (defaults to `['blog']`)

**Parameters** **account** (*str*) – (optional) the account to allow access to (defaults to `default_account`)

**getSimilarAccountNames**(*limit=5*)
Returns limit similar accounts with name as array

**get_account_bandwidth**(*bandwidth_type=1*, *account=None*)

**get_account_history**(*index*, *limit*, *order=-1*, *start=None*, *stop=None*, *use_block_num=True*, *only_ops=[]*, *exclude_ops=[]*, *raw_output=False*)
Returns a generator for individual account transactions. This call can be used in a `for` loop. :param int index: first number of transactions to

return

**Parameters**

- **limit** (*int*) – limit number of transactions to return
- **start** (*int/datetime*) – start number/date of transactions to return (*optional*)

- **stop** (*int/datetime*) – stop number/date of transactions to return (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)
- **batch_size** (*int*) – internal api call batch size (*optional*)
- **order** (*int*) – 1 for chronological, -1 for reverse order
- **raw_output** (*bool*) – if False, the output is a dict, which includes all values. Otherwise, the output is list.

... note:: only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in beembase.operationids.ops. Example: ['transfer', 'vote']

**get_account_votes**(*account=None*)
Returns all votes that the account has done

**get_balance**(*balances*, *symbol*)
Obtain the balance of a specific Asset. This call returns instances of `steem.amount.Amount`.

**get_balances**()

**get_bandwidth**()
Returns used and allocated bandwidth

**get_blog**(*entryId=0*, *limit=100*, *raw_data=False*, *account=None*)

**get_blog_account**(*account=None*)

**get_blog_entries**(*entryId=0*, *limit=100*, *raw_data=False*, *account=None*)

**get_conversion_requests**(*account=None*)
get_owner_history

**get_curation_reward**(*days=7*)
Returns the curation reward of the last *days* days

**Parameters** **days** (*int*) – limit number of days to be included int the return value

**get_feed**(*entryId=0*, *limit=100*, *raw_data=False*, *account=None*)

**get_follow_count**(*account=None*)

**get_followers**(*raw_name_list=True*)
Returns the account followers as list

**get_following**(*raw_name_list=True*)
Returns who the account is following as list

**get_muters**(*raw_name_list=True*)
Returns the account muters as list

**get_mutings**(*raw_name_list=True*)
Returns who the account is muting as list

**get_owner_history**(*account=None*)

**get_recharge_time**(*voting_power_goal=100*)
Returns the account voting power recharge time in minutes

**get_recharge_time_str**(*voting_power_goal=100*)
Returns the account recharge time

**get_recharge_timedelta**(*voting_power_goal=100*)
Returns the account voting power recharge time as timedelta object

**get_recovery_request**(*account=None*)

**get_reputation**()
Returns the account reputation

**get_steem_power**(*onlyOwnSP=False*)
Returns the account steem power

**get_vote**(*comment*)
Returns a vote if the account has already voted for comment.

**Parameters** **comment** (*str/Comment*) – can be a Comment object or a authorpermlink

**get_voting_power**(*with_regeneration=True*)
Returns the account voting power

**get_voting_value_SBD**(*voting_weight=100*, *voting_power=None*, *steem_power=None*)
Returns the account voting value in SBD

**get_withdraw_routes**(*account=None*)
Returns withdraw_routes

**has_voted**(*comment*)
Returns if the account has already voted for comment

**Parameters** **comment** (*str/Comment*) – can be a Comment object or a authorpermlink

**history**(*start=None*, *stop=None*, *use_block_num=True*, *only_ops=[]*, *exclude_ops=[]*, *batch_size=1000*, *raw_output=False*)
Returns a generator for individual account transactions. The earlist operation will be first. This call can be used in a `for` loop.

**Parameters**

- **start** (*int/datetime*) – start number/date of transactions to return (*optional*)
- **stop** (*int/datetime*) – stop number/date of transactions to return (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)
- **batch_size** (*int*) – internal api call batch size (*optional*)
- **raw_output** (*bool*) – if False, the output is a dict, which includes all values. Otherwise, the output is list.

**... note::** only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in beembase.operationids.ops. Example: ['transfer', 'vote']

**Example::** from beem.account import Account from beem.blockchain import Blockchain from datetime import datetime acc = Account("test") max_op_count = acc.virtual_op_count() # Returns the 100 latest operations for h in acc.history(start=max_op_count-100, stop=max_op_count, use_block_num=False):

print(h)

b = Blockchain() max_block = b.get_current_block_num() # Returns the account operation inside the last 100 block. This can be empty. for h in acc.history(start=max_block-100, stop=max_block, use_block_num=True):

print(h)

start_time = datetime(2018, 3, 1, 0, 0, 0) stop_time = datetime(2018, 4, 1, 0, 0, 0) # Returns the account operation from 1.4.2018 back to 1.3.2018 for h in acc.history(start=start_time, stop=stop_time):

print(h)

**history_reverse**(*start=None*, *stop=None*, *use_block_num=True*, *only_ops=[]*, *exclude_ops=[]*, *batch_size=1000*, *raw_output=False*)

Returns a generator for individual account transactions. The latest operation will be first. This call can be used in a `for` loop.

**Parameters**

- **start** (*int/datetime*) – start number/date of transactions to return. If negative the virtual_op_count is added. (*optional*)
- **stop** (*int/datetime*) – stop number/date of transactions to return. If negative the virtual_op_count is added. (*optional*)
- **use_block_num** (*bool*) – if true, start and stop are block numbers, otherwise virtual OP count numbers.
- **only_ops** (*array*) – Limit generator by these operations (*optional*)
- **exclude_ops** (*array*) – Exclude thse operations from generator (*optional*)
- **batch_size** (*int*) – internal api call batch size (*optional*)
- **raw_output** (*bool*) – if False, the output is a dict, which includes all values. Otherwise, the output is list.

**... note::** only_ops and exclude_ops takes an array of strings: The full list of operation ID's can be found in beembase.operationids.ops. Example: ['transfer', 'vote']

**Example::** from beem.account import Account from beem.blockchain import Blockchain from datetime import datetime acc = Account("test") max_op_count = acc.virtual_op_count() # Returns the 100 latest operations for h in acc.history_reverse(start=max_op_count, stop=max_op_count-100, use_block_num=False):

print(h)

b = Blockchain() max_block = b.get_current_block_num() # Returns the account operation inside the last 100 block. This can be empty. for h in acc.history_reverse(start=max_block, stop=max_block-100, use_block_num=True):

print(h)

start_time = datetime(2018, 4, 1, 0, 0, 0) stop_time = datetime(2018, 3, 1, 0, 0, 0) # Returns the account operation from 1.4.2018 back to 1.3.2018 for h in acc.history_reverse(start=start_time, stop=stop_time):

print(h)

**interest**()

Caluclate interest for an account :param str account: Account name to get interest for

**is_fully_loaded**

Is this instance fully loaded / e.g. all data available?

**json**()

**mute**(*mute*, *account=None*)
Mute another account :param str mute: Mute this account :param str account: (optional) the account to allow access

to (defaults to `default_account`)

**name**
Returns the account name

**print_info**(*force_refresh=False*, *return_str=False*, *use_table=False*, ***kwargs*)
Prints import information about the account

**profile**
Returns the account profile

**refresh**()
Refresh/Obtain an account's data from the API server

**rep**
Returns the account reputation

**reward_balances**

**saving_balances**

**set_withdraw_vesting_route**(*to*, *percentage=100*, *account=None*, *auto_vest=False*)
Set up a vesting withdraw route. When vesting shares are withdrawn, they will be routed to these accounts based on the specified weights. :param str to: Recipient of the vesting withdrawal :param float percentage: The percent of the withdraw to go

to the 'to' account.

**Parameters**

- **account** (*str*) – (optional) the vesting account
- **auto_vest** (*bool*) – Set to true if the from account should receive the VESTS as VESTS, or false if it should receive them as STEEM. (defaults to `False`)

**sp**

**total_balances**

**transfer**(*to*, *amount*, *asset*, *memo=''*, *account=None*, ***kwargs*)
Transfer an asset to another account.

**Parameters**

- **to** (*str*) – Recipient
- **amount** (*float*) – Amount to transfer
- **asset** (*str*) – Asset to transfer
- **memo** (*str*) – (optional) Memo, may begin with # for encrypted messaging
- **account** (*str*) – (optional) the source account for the transfer if not `default_account`

**transfer_from_savings**(*amount*, *asset*, *memo*, *request_id=None*, *to=None*, *account=None*)
Withdraw SBD or STEEM from 'savings' account. :param float amount: STEEM or SBD amount :param float asset: 'STEEM' or 'SBD' :param str memo: (optional) Memo :param str request_id: (optional) identifier for tracking or cancelling the withdrawal :param str to: (optional) the source account for the

transfer if not default_account :param str account: (optional) the source account for the transfer if not default_account

**transfer_to_savings**(*amount*, *asset*, *memo*, *to=None*, *account=None*)
Transfer SBD or STEEM into a 'savings' account. :param float amount: STEEM or SBD amount :param float asset: 'STEEM' or 'SBD' :param str memo: (optional) Memo :param str to: (optional) the source account for the transfer if not default_account :param str account: (optional) the source account for the transfer if not default_account

**transfer_to_vesting**(*amount*, *to=None*, *account=None*, ***kwargs*)
Vest STEEM

**Parameters**

- **amount** (*float*) – Amount to transfer
- **to** (*str*) – Recipient (optional) if not set equal to account
- **account** (*str*) – (optional) the source account for the transfer if not default_account

**type_id = 2**

**unfollow**(*unfollow*, *account=None*)
Unfollow/Unmute another account's blog :param str unfollow: Unfollow/Unmute this account :param str account: (optional) the account to allow access

to (defaults to default_account)

**update_account_profile**(*profile*, *account=None*)
Update an account's meta data (json_meta) :param dict json: The meta data to use (i.e. use Profile() from

account.py)

**Parameters** **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

**update_memo_key**(*key*, *account=None*, ***kwargs*)
Update an account's memo public key

This method does **not** add any private keys to your wallet but merely changes the memo public key.

**Parameters**

- **key** (*str*) – New memo public key
- **account** (*str*) – (optional) the account to allow access to (defaults to default_account)

**verify_account_authority**(*keys*, *account=None*)

**virtual_op_count**(*until=None*)
Returns the number of individual account transactions

**vp**

**withdraw_vesting**(*amount*, *account=None*)
Withdraw VESTS from the vesting account. :param float amount: number of VESTS to withdraw over a period of 104 weeks :param str account: (optional) the source account for the transfer if not default_account

**class** beem.account.**Accounts**(*name_list*, *batch_limit=100*, *steem_instance=None*)
Bases: *beem.account.AccountsObject*

Obtain a list of accounts

**Parameters** **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.account.**AccountsObject**
Bases: list

**printAsTable**()

**print_summarize_table**(*tag_type='Follower'*, *return_str=False*, ***kwargs*)

### beem.aes module

**class** beem.aes.**AESCipher**(*key*)
Bases: object

A classical AES Cipher. Can use any size of data and any size of password thanks to padding. Also ensure the coherence and the type of the data with a unicode to byte converter.

**decrypt**(*enc*)

**encrypt**(*raw*)

**static str_to_bytes**(*data*)

### beem.amount module

**class** beem.amount.**Amount**(*amount*, *asset=None*, *new_appbase_format=False*, *steem_instance=None*)
Bases: dict

This class deals with Amounts of any asset to simplify dealing with the tuple:

```
(amount, asset)
```

**Parameters**

- **args** (*list*) – Allows to deal with different representations of an amount
- **amount** (*float*) – Let's create an instance with a specific amount
- **asset** (*str*) – Let's you create an instance with a specific asset (symbol)
- **steem_instance** (*steem.steem.Steem*) – Steem instance

**Returns** All data required to represent an Amount/Asset

**Return type** dict

**Raises** **ValueError** – if the data provided is not recognized

Way to obtain a proper instance:

- `args` can be a string, e.g.: "1 SBD"
- `args` can be a dictionary containing `amount` and `asset_id`
- `args` can be a dictionary containing `amount` and `asset`
- `args` can be a list of a `float` and `str` (symbol)
- `args` can be a list of a `float` and a *beem.asset.Asset*

- amount and asset are defined manually

An instance is a dictionary and comes with the following keys:

- amount (float)
- symbol (str)
- asset (instance of *beem.asset.Asset*)

Instances of this class can be used in regular mathematical expressions (+-*/%) such as:

```
from beem.amount import Amount
from beem.asset import Asset
a = Amount("1 STEEM")
b = Amount(1, "STEEM")
c = Amount("20", Asset("STEEM"))
a + b
a * 2
a += b
a /= 2.0
```

**amount**
Returns the amount as float

**asset**
Returns the asset as instance of steem.asset.Asset

**copy**()
Copy the instance and make sure not to use a reference

**json**()

**symbol**
Returns the symbol of the asset

**tuple**()

### beem.asset module

**class** beem.asset.**Asset**(*asset*, *lazy=False*, *full=False*, *steem_instance=None*)
Bases: beem.blockchainobject.BlockchainObject

Deals with Assets of the network.

**Parameters**

- **Asset** (*str*) – Symbol name or object id of an asset
- **lazy** (*bool*) – Lazy loading
- **full** (*bool*) – Also obtain bitasset-data and dynamic asset dat
- **steem_instance** (beem.steem.Steem) – Steem instance

**Returns** All data of an asset

**Return type** dict

**Note:** This class comes with its own caching function to reduce the load on the API server. Instances of this class can be refreshed with Asset.refresh().

**asset**

**precision**

**refresh**()
Refresh the data from the API server

**symbol**

**type_id = 3**

### beem.steem module

**class** beem.steem.**Steem**(*node=”*, *rpcuser=None*, *rpcpassword=None*, *debug=False*, *data_refresh_time_seconds=900*, ***kwargs*)
Bases: object

Connect to the Steem network.

**Parameters**

- **node** (*str*) – Node to connect to *(optional)*
- **rpcuser** (*str*) – RPC user *(optional)*
- **rpcpassword** (*str*) – RPC password *(optional)*
- **nobroadcast** (*bool*) – Do **not** broadcast a transaction! *(optional)*
- **debug** (*bool*) – Enable Debugging *(optional)*
- **keys** (*array,dict,string*) – Predefine the wif keys to shortcut the wallet database *(optional)*
- **wif** (*array,dict,string*) – Predefine the wif keys to shortcut the wallet database *(optional)*
- **offline** (*bool*) – Boolean to prevent connecting to network (defaults to False) *(optional)*
- **expiration** (*int*) – Delay in seconds until transactions are supposed to expire *(optional)*
- **blocking** (*str*) – Wait for broadcasted transactions to be included in a block and return full transaction (can be “head” or “irrversible”)
- **bundle** (*bool*) – Do not broadcast transactions right away, but allow to bundle operations *(optional)*
- **appbase** (*bool*) – Use the new appbase rpc protocol on nodes with version 0.19.4 or higher. The settings has no effect on nodes with version of 0.19.3 or lower.
- **num_retries** (*int*) – Set the maximum number of reconnects to the nodes before NumRetriesReached is raised. Disabled for -1. (default is -1)
- **num_retries_call** (*int*) – Repeat num_retries_call times a rpc call on node error (default is 5)
- **timeout** (*int*) – Timeout setting for https nodes (default is 60)

Three wallet operation modes are possible:

- **Wallet Database**: Here, the steemlibs load the keys from the locally stored wallet SQLite database (see storage.py). To use this mode, simply call Steem() without the keys parameter

- **Providing Keys**: Here, you can provide the keys for your accounts manually. All you need to do is add the wif keys for the accounts you want to use as a simple array using the `keys` parameter to `Steem()`.
- **Force keys**: This more is for advanced users and requires that you know what you are doing. Here, the `keys` parameter is a dictionary that overwrite the `active`, `owner`, `posting` or `memo` keys for any account. This mode is only used for *foreign* signatures!

If no node is provided, it will connect to default nodes of http://geo.steem.pl. Default settings can be changed with:

```
steem = Steem(<host>)
```

where `<host>` starts with `https://`, `ws://` or `wss://`.

The purpose of this class it to simplify interaction with Steem.

The idea is to have a class that allows to do this:

```
from beem import Steem
steem = Steem()
print(steem.info())
```

This class also deals with edits, votes and reading content.

**broadcast**(*tx=None*)
Broadcast a transaction to the Steem network

**Parameters** **tx** (*tx*) – Signed transaction to broadcast

**chain_params**

**clear**()

**comment_options**(*options*, *identifier*, *account=None*)
Set the comment options :param str identifier: Post identifier :param dict options: The options to define. :param str account: (optional) the account to allow access

to (defaults to `default_account`)

**For the options, you have these defaults:::**

{ "author": "", "permlink": "", "max_accepted_payout": "1000000.000 SBD", "percent_steem_dollars": 10000, "allow_votes": True, "allow_curation_rewards": True,

}

**connect**(*node=''*, *rpcuser=''*, *rpcpassword=''*, ***kwargs*)
Connect to Steem network (internal use only)

**create_account**(*account_name*, *creator=None*, *owner_key=None*, *active_key=None*, *memo_key=None*, *posting_key=None*, *password=None*, *additional_owner_keys=[]*, *additional_active_keys=[]*, *additional_posting_keys=[]*, *additional_owner_accounts=[]*, *additional_active_accounts=[]*, *additional_posting_accounts=[]*, *storekeys=True*, *store_owner_key=False*, *json_meta=None*, *delegation_fee_steem='0 STEEM'*, ***kwargs*)
Create new account on Steem

The brainkey/password can be used to recover all generated keys (see *beemgraphenebase.account* for more details.

By default, this call will use `default_account` to register a new name `account_name` with all keys being derived from a new brain key that will be returned. The corresponding keys will automatically be installed in the wallet.

**Warning:** Don't call this method unless you know what you are doing! Be sure to understand what this method does and where to find the private keys for your account.

**Note:** Please note that this imports private keys (if password is present) into the wallet by default when nobroadcast is set to False. However, it **does not import the owner key** for security reasons by default. If you set store_owner_key to True, the owner key is stored. Do NOT expect to be able to recover it from the wallet if you lose your password!

**Note:** Account creations cost a fee that is defined by the network. If you create an account, you will need to pay for that fee! **You can partially pay that fee by delegating VESTS.** To pay the fee in full in STEEM, leave `delegation_fee_steem` set to `0 STEEM` (Default). To pay the fee partially in STEEM, partially with delegated VESTS, set `delegation_fee_steem` to a value greater than `1 STEEM`. *Required VESTS will be calculated automatically.* To pay the fee with maximum amount of delegation, set `delegation_fee_steem` to `1 STEEM`. *Required VESTS will be calculated automatically.*

**Parameters**

- **account_name** (*str*) – (**required**) new account name
- **json_meta** (*str*) – Optional meta data for the account
- **owner_key** (*str*) – Main owner key
- **active_key** (*str*) – Main active key
- **posting_key** (*str*) – Main posting key
- **memo_key** (*str*) – Main memo_key
- **password** (*str*) – Alternatively to providing keys, one can provide a password from which the keys will be derived
- **additional_owner_keys** (*array*) – Additional owner public keys
- **additional_active_keys** (*array*) – Additional active public keys
- **additional_posting_keys** (*array*) – Additional posting public keys
- **additional_owner_accounts** (*array*) – Additional owner account names
- **additional_active_accounts** (*array*) – Additional acctive account names
- **storekeys** (*bool*) – Store new keys in the wallet (default: `True`)
- **delegation_fee_steem** – If set, *creator* pay a fee of this amount, and delegate the rest with VESTS (calculated automatically). Minimum: 1 STEEM. If left to 0 (Default), full fee is paid without VESTS delegation.
- **creator** (*str*) – which account should pay the registration fee (defaults to `default_account`)

**Raises** ***AccountExistsException*** – if the account already exists on the blockchain

**custom_json**(*id*, *json_data*, *required_auths=[]*, *required_posting_auths=[]*)
Create a custom json operation :param str id: identifier for the custom json (max length 32 bytes) :param json json_data: the json data to put into the custom_json

operation

**Parameters**

- **required_auths** (*list*) – (optional) required auths
- **required_posting_auths** (*list*) – (optional) posting auths

**finalizeOp**(*ops*, *account*, *permission*, ***kwargs*)
This method obtains the required private keys if present in the wallet, finalizes the transaction, signs it and broadacasts it

**Parameters**

- **ops** (*operation*) – The operation (or list of operaions) to broadcast
- **account** (*operation*) – The account that authorizes the operation
- **permission** (*string*) – The required permission for signing (active, owner, posting)
- **append_to** (*object*) – This allows to provide an instance of ProposalsBuilder (see `steem.new_proposal()`) or TransactionBuilder (see `steem.new_tx()`) to specify where to put a specific operation.

**... note:: append_to is exposed to every method used in the** Steem class

... note:

```
If ``ops`` is a list of operation, they all need to be
signable by the same key! Thus, you cannot combine ops
that require active permission with ops that require
posting permission. Neither can you use different
accounts for different operations!
```

**... note:: This uses beem.txbuffer as instance of** *beem.transactionbuilder.TransactionBuilder*. You may want to use your own txbuffer

**get_block_interval**()
Returns the block intervall in seconds

**get_blockchain_version**()
Returns the blockchain version

**get_chain_properties**(*use_stored_data=True*)
Return witness elected chain properties

**::**

**{'account_creation_fee': '30.000 STEEM',** 'maximum_block_size': 65536, 'sbd_interest_rate': 250}

**get_config**(*use_stored_data=True*)
Returns internal chain configuration.

**get_current_median_history**(*use_stored_data=True*)
Returns the current median price :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**get_default_nodes**()
Returns the default nodes

**get_dynamic_global_properties**(*use_stored_data=True*)
This call returns the *dynamic global properties* :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**get_feed_history**(*use_stored_data=True*)
Returns the feed_history :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**get_hardfork_properties**(*use_stored_data=True*)
Returns Hardfork and live_time of the hardfork :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**get_median_price**()
Returns the current median history price as Price

**get_network**(*use_stored_data=True*)
Identify the network :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**Returns** Network parameters

**Return type** dict

**get_reserve_ratio**(*use_stored_data=True*)
This call returns the *dynamic global properties* :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**get_reward_funds**(*use_stored_data=True*)
Get details for a reward fund. :param bool use_stored_data: if True, stored data will be returned. If stored data are empty or old, refresh_data() is used.

**get_sbd_per_rshares**()
Returns the current rshares to SBD ratio

**get_steem_per_mvest**(*time_stamp=None*)
Returns the current mvest to steem ratio

**get_witness_schedule**(*use_stored_data=True*)
Return witness elected chain properties

**info**()
Returns the global properties

**is_connected**()
Returns if rpc is connected

**move_current_node_to_front**()
Returns the default node list, until the first entry is equal to the current working node url

**newWallet**(*pwd*)
Create a new wallet. This method is basically only calls `beem.wallet.create()`.

**Parameters** **pwd** (*str*) – Password to use for the new wallet

**Raises** ***beem.exceptions.WalletExists*** – if there is already a wallet created

**new_tx**(**args*, ***kwargs*)
Let's obtain a new txbuffer

**Returns int txid** id of the new txbuffer

**post**(*title*, *body*, *author=None*, *permlink=None*, *reply_identifier=None*, *json_metadata=None*, *comment_options=None*, *community=None*, *app=None*, *tags=None*, *beneficiaries=None*, *self_vote=False*)

Create a new post. If this post is intended as a reply/comment, *reply_identifier* needs to be set with the identifier of the parent post/comment (eg. *@author/permlink*). Optionally you can also set json_metadata, comment_options and upvote the newly created post as an author. Setting category, tags or community will override the values provided in json_metadata and/or comment_options where appropriate. Args: title (str): Title of the post body (str): Body of the post/comment author (str): Account are you posting from permlink (str): Manually set the permlink (defaults to None).

If left empty, it will be derived from title automatically.

**reply_identifier (str): Identifier of the parent post/comment (only** if this post is a reply/comment).

**json_metadata (str, dict): JSON meta object that can be attached to** the post.

**comment_options (str, dict): JSON options object that can be** attached to the post.

**Example::**

**comment_options = {** 'max_accepted_payout': '1000000.000 SBD', 'percent_steem_dollars': 10000, 'allow_votes': True, 'allow_curation_rewards': True, 'extensions': [[0, {

**'beneficiaries': [** {'account': 'account1', 'weight': 5000}, {'account': 'account2', 'weight': 5000},

]}

]]

}

**community (str): (Optional) Name of the community we are posting** into. This will also override the community specified in *json_metadata*.

**app (str): (Optional) Name of the app which are used for posting** when not set, beem/<version> is used

**tags (str, list): (Optional) A list of tags (5 max) to go with the** post. This will also override the tags specified in *json_metadata*. The first tag will be used as a 'category'. If provided as a string, it should be space separated.

**beneficiaries (list of dicts): (Optional) A list of beneficiaries** for posting reward distribution. This argument overrides beneficiaries as specified in *comment_options*.

For example, if we would like to split rewards between account1 and account2:

```
beneficiaries = [
    {'account': 'account1', 'weight': 5000},
    {'account': 'account2', 'weight': 5000}
]
```

**self_vote (bool): (Optional) Upvote the post as author, right after** posting.

**prefix**

**refresh_data**(*force_refresh=False*, *data_refresh_time_seconds=None*)

Read and stores steem blockchain parameters If the last data refresh is older than data_refresh_time_seconds, data will be refreshed

**param bool force_refresh** if True, data are forced to refreshed

**param float data_refresh_time_seconds** set a new minimal refresh time in seconds

**rshares_to_sbd**(*rshares*)
Calculates the SBD amount of a vote

**rshares_to_vote_pct**(*rshares*, *steem_power=None*, *vests=None*, *voting_power=10000*)
Obtain the voting percentage for a desired rshares value for a given Steem Power or vesting shares and voting_power Give either steem_power or vests, not both. When the output is greater than 10000, the given rshares are to high

Returns the voting participation (100% = 10000)

**Parameters**

- **rshares** (*number*) – desired rshares value
- **steem_power** (*number*) – Steem Power
- **vests** (*number*) – vesting shares
- **voting_power** (*int*) – voting power (100% = 10000)

**set_default_account**(*account*)
Set the default account to be used

**set_default_nodes**(*nodes*)
Set the default nodes to be used

**set_default_vote_weight**(*vote_weight*)
Set the default vote weight to be used

**set_password_storage**(*password_storage*)
Set the password storage mode.

When set to “no”, the password has to provided everytime. When set to “environment” the password is taken from the

UNLOCK variable

When set to “keyring” the password is taken from the python keyring module. A wallet password can be stored with python -m keyring set beem wallet password :param str password_storage: can be “no”,

“keyring” or “environment”

**sign**(*tx=None*, *wifs=[]*)
Sign a provided transaction witht he provided key(s)

**Parameters**

- **tx** (*dict*) – The transaction to be signed and returned
- **wifs** (*string*) – One or many wif keys to use for signing a transaction. If not present, the keys will be loaded from the wallet as defined in “missing_signatures” key of the transactions.

**sp_to_rshares**(*steem_power*, *voting_power=10000*, *vote_pct=10000*)
Obtain the r-shares from Steem power :param number steem_power: Steem Power :param int voting_power: voting power (100% = 10000) :param int vote_pct: voting participation (100% = 10000)

**sp_to_sbd**(*sp*, *voting_power=10000*, *vote_pct=10000*)
Obtain the resulting sbd amount from Steem power :param number steem_power: Steem Power :param int voting_power: voting power (100% = 10000) :param int vote_pct: voting participation (100% = 10000)

**sp_to_vests**(*sp*, *timestamp=None*)

**tx**()
Returns the default transaction buffer

**txbuffer**
Returns the currently active tx buffer

**unlock**(**args*, ***kwargs*)
Unlock the internal wallet

**vests_to_rshares**(*vests*, *voting_power=10000*, *vote_pct=10000*)
Obtain the r-shares from vests :param number vests: vesting shares :param int voting_power: voting power (100% = 10000) :param int vote_pct: voting participation (100% = 10000)

**vests_to_sbd**(*vests*, *voting_power=10000*, *vote_pct=10000*)
Obtain the resulting sbd voting amount from vests :param number vests: vesting shares :param int voting_power: voting power (100% = 10000) :param int vote_pct: voting participation (100% = 10000)

**vests_to_sp**(*vests*, *timestamp=None*)

**witness_update**(*signing_key*, *url*, *props*, *account=None*)
Creates/updates a witness :param pubkey signing_key: Signing key :param str url: URL :param dict props: Properties :param str account: (optional) witness account name

**Properties:::**

{ "account_creation_fee": x, "maximum_block_size": x, "sbd_interest_rate": x,

}

## beem.block module

**class** beem.block.**Block**(*data*, *klass=None*, *space_id=1*, *object_id=None*, *lazy=False*, *use_cache=True*, *id_item=None*, *steem_instance=None*, **args*, ***kwargs*)
Bases: beem.blockchainobject.BlockchainObject

Read a single block from the chain

**Parameters**

- **block** (*int*) – block number
- **steem_instance** (beem.steem.Steem) – Steem instance
- **lazy** (*bool*) – Use lazy loading

Instances of this class are dictionaries that come with additional methods (see below) that allow dealing with a block and it's corresponding functions.

Additionally to the block data, the block number is stored as self["id"] or self.identifier.

```
from beem.block import Block
block = Block(1)
print(block)
```

**Note:** This class comes with its own caching function to reduce the load on the API server. Instances of this class can be refreshed with Account.refresh().

**block_num**
Returns the block number

**ops**()
Returns all block operations

**ops_statistics**(*add_to_ops_stat=None*)
Retuns a statistic with the occurance of the different operation types

**refresh**()
Even though blocks never change, you freshly obtain its contents from an API with this method

**time**()
Return a datatime instance for the timestamp of this block

**class** beem.block.**BlockHeader**(*data*, *klass=None*, *space_id=1*, *object_id=None*, *lazy=False*, *use_cache=True*, *id_item=None*, *steem_instance=None*, **args*, ***kwargs*)
Bases: beem.blockchainobject.BlockchainObject

**block_num**
Retuns the block number

**refresh**()
Even though blocks never change, you freshly obtain its contents from an API with this method

**time**()
Return a datatime instance for the timestamp of this block

### beem.blockchain module

**class** beem.blockchain.**Blockchain**(*steem_instance=None*, *mode='irreversible'*, *max_block_wait_repetition=None*, *data_refresh_time_seconds=900*)
Bases: object

This class allows to access the blockchain and read data from it

**Parameters**

- **steem_instance** (beem.steem.Steem) – Steem instance
- **mode** (*str*) – (default) Irreversible block (irreversible) or actual head block (head)
- **max_block_wait_repetition** (*int*) – (default) 3 maximum wait time for next block is max_block_wait_repetition * block_interval

This class let's you deal with blockchain related data and methods. Read blockchain related data: .. code-block:: python

from beem.blockchain import Blockchain chain = Blockchain()

Read current block and blockchain info .. code-block:: python

print(chain.get_current_block()) print(chain.steem.info())

Monitor for new blocks .. .. code-block:: python

**for block in chain.blocks():** print(block)

or each operation individually: .. code-block:: python

**for operations in chain.ops():** print(operations)

**awaitTxConfirmation**(*transaction*, *limit=10*)
Returns the transaction as seen by the blockchain after being included into a block

**Note:** If you want instant confirmation, you need to instantiate class:*beem.blockchain.Blockchain* with `mode="head"`, otherwise, the call will wait until confirmed in an irreversible block.

**Note:** This method returns once the blockchain has included a transaction with the **same signature**. Even though the signature is not usually used to identify a transaction, it still cannot be forfeited and is derived from the transaction contented and thus identifies a transaction uniquely.

**block_time**(*block_num*)
Returns a datetime of the block with the given block number.

**Parameters** **block_num** (*int*) – Block number

**block_timestamp**(*block_num*)
Returns the timestamp of the block with the given block number.

**Parameters** **block_num** (*int*) – Block number

**blocks**(*start=None*, *stop=None*, *max_batch_size=None*, *threading=False*, *thread_num=8*)
Yields blocks starting from `start`.

**Parameters**

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block
- **mode** (*str*) – We here have the choice between "head" (the last block) and "irreversible" (the block that is confirmed by 2/3 of all block producers and is thus irreversible)

**get_all_accounts**(*start=''*, *stop=''*, *steps=1000.0*, *limit=-1*, ***kwargs*)
Yields account names between start and stop.

**Parameters**

- **start** (*str*) – Start at this account name
- **stop** (*str*) – Stop at this account name
- **steps** (*int*) – Obtain `steps` ret with a single call from RPC

**get_current_block**()
This call returns the current block

**Note:** The block number returned depends on the `mode` used when instanciating from this class.

**get_current_block_num**()
This call returns the current block number

**Note:** The block number returned depends on the `mode` used when instanciating from this class.

**get_estimated_block_num**(*date*, *estimateForwards=False*, *accurate=True*)
This call estimates the block number based on a given date

**Parameters** **date** (*datetime*) – block time for which a block number is estimated

**Note:** The block number returned depends on the `mode` used when instanciating from this class.

**static hash_op**(*event*)
This method generates a hash of blockchain operation.

**is_irreversible_mode**()

**ops**(*start=None*, *stop=None*, ***kwargs*)
Yields all operations (including virtual operations) starting from `start`.

**Parameters**

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block
- **mode** (*str*) – We here have the choice between "head" (the last block) and "irreversible" (the block that is confirmed by 2/3 of all block producers and is thus irreversible)
- **only_virtual_ops** (*bool*) – Only yield virtual operations

This call returns a list that only carries one operation and its type!

**ops_statistics**(*start*, *stop=None*, *add_to_ops_stat=None*, *verbose=False*)
Generates a statistics for all operations (including virtual operations) starting from `start`.

**Parameters**

- **start** (*int*) – Starting block
- **stop** (*int*) – Stop at this block, if set to None, the current_block_num is taken

:param dict add_to_ops_stat, if set, the result is added to add_to_ops_stat :param bool verbose, if True, the current block number and timestamp is printed This call returns a dict with all possible operations and their occurence.

**stream**(*opNames=[]*, **args*, ***kwargs*)
Yield specific operations (e.g. comments) only

**Parameters**

- **opNames** (*array*) – List of operations to filter for
- **start** (*int*) – Start at this block
- **stop** (*int*) – Stop at this block
- **mode** (*str*) – We here have the choice between "head" (the last block) and "irreversible" (the block that is confirmed by 2/3 of all block producers and is thus irreversible)

The dict output is formated such that `type` caries the operation type, timestamp and block_num are taken from the block the operation was stored in and the other key depend on the actualy operation.

**wait_for_and_get_block**(*block_number*, *blocks_waiting_for=None*, *last_fetched_block_num=None*)
Get the desired block from the chain, if the current head block is smaller (for both head and irreversible) then we wait, but a maxmimum of blocks_waiting_for * max_block_wait_repetition time before failure. :param int block_number: desired block number :param int blocks_waiting_for: (default) difference between block_number and current head

how many blocks we are willing to wait, positive int

### beem.comment module

**class** beem.comment.**Comment**(*authorperm*, *full=True*, *lazy=False*, *steem_instance=None*)
Bases: beem.blockchainobject.BlockchainObject

Read data about a Comment/Post in the chain

**Parameters**

- **authorperm** (*str*) – perm link to post/comment
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**author**

**authorperm**

**body**

**category**

**delete**(*account=None*, *identifier=None*)
Delete an existing post/comment :param str identifier: Identifier for the post to upvote Takes

the form @author/permlink

**Parameters** **account** (*str*) – Voter to use for voting. (Optional)

If voter is not defines, the default_account will be taken or a ValueError will be raised

**downvote**(*weight=-100*, *voter=None*)
Downvote the post :param float weight: (optional) Weight for posting (-100.0 - +100.0) defaults to -100.0 :param str voter: (optional) Voting account

**edit**(*body*, *meta=None*, *replace=False*)
Edit an existing post :param str body: Body of the reply :param json meta: JSON meta object that can be attached to the

post. (optional)

**Parameters** **replace** (*bool*) – Instead of calculating a *diff*, replace the post entirely (defaults to False)

**get_reblogged_by**(*identifier=None*)

**get_votes**()

**id**

**is_comment**()
Retuns True if post is a comment

**is_main_post**()
Retuns True if main post, and False if this is a comment (reply).

**json**()

**json_metadata**

**parent_author**

**parent_permlink**

**permlink**

**refresh**()

**reply**(*body*, *title=''*, *author=''*, *meta=None*)
Reply to an existing post :param str body: Body of the reply :param str title: Title of the reply post :param str author: Author of reply (optional) if not provided

default_user will be used, if present, else a ValueError will be raised.

**Parameters** **meta** (*json*) – JSON meta object that can be attached to the post. (optional)

**resteem**(*identifier=None*, *account=None*)
Resteem a post :param str identifier: post identifier (@<account>/<permlink>) :param str account: (optional) the account to allow access

to (defaults to default_account)

**title**

**type_id = 8**

**upvote**(*weight=100*, *voter=None*)
Upvote the post :param float weight: (optional) Weight for posting (-100.0 - +100.0) defaults to +100.0 :param str voter: (optional) Voting account

**vote**(*weight*, *account=None*, *identifier=None*, ***kwargs*)
Vote for a post :param str identifier: Identifier for the post to upvote Takes

the form @author/permlink

**Parameters**

- **weight** (*float*) – Voting weight. Range: -100.0 - +100.0. May not be 0.0
- **account** (*str*) – Voter to use for voting. (Optional)

If voter is not defines, the default_account will be taken or a ValueError will be raised

**class** beem.comment.**RecentByPath**(*path='promoted'*, *category=None*, *steem_instance=None*)
Bases: list

Obtain a list of votes for an account

**Parameters**

- **account** (*str*) – Account name
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.comment.**RecentReplies**(*author*, *skip_own=True*, *steem_instance=None*)
Bases: list

Obtain a list of recent replies

**Parameters**

- **author** (*str*) – author
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

### beem.discussions module

**class** beem.discussions.**Comment_discussions_by_payout**(*discussion_query*, *steem_instance=None*)

Bases: list

get_comment_discussions_by_payout

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_active**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_active

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_blog**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_blog

:param str discussion_query, tag musst be set to a username :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_cashout**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_cashout. This query seems to be broken at the moment. The output is always empty.

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_children**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_children

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_comments**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_comments

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_created**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_created

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_feed**(*discussion_query*, *steem_instance=None*)

Bases: list

get_discussions_by_feed

:param str discussion_query, tag musst be set to a username :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_hot**(*discussion_query*, *steem_instance=None*)
Bases: list

get_discussions_by_hot

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_promoted**(*discussion_query*, *steem_instance=None*)
Bases: list

get_discussions_by_promoted

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_trending**(*discussion_query*, *steem_instance=None*)
Bases: list

get_discussions_by_trending

:param Query discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Discussions_by_votes**(*discussion_query*, *steem_instance=None*)
Bases: list

get_discussions_by_votes

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Post_discussions_by_payout**(*discussion_query*, *steem_instance=None*)
Bases: list

get_post_discussions_by_payout

:param str discussion_query :param steem steem_instance: Steem() instance to use when accesing a RPC

**class** beem.discussions.**Query**(*limit=0*, *tag=''*, *truncate_body=0*, *filter_tags=[]*, *select_authors=[]*, *select_tags=[]*, *start_author=None*, *start_permlink=None*, *parent_author=None*, *parent_permlink=None*)
Bases: dict

:param int limit :param str tag :param int truncate_body :param array filter_tags :param array select_authors :param array select_tags :param str start_author :param str start_permlink :param str parent_author :param str parent_permlink

### beem.exceptions module

**exception** beem.exceptions.**AccountDoesNotExistsException**
Bases: Exception

The account does not exist

**exception** beem.exceptions.**AccountExistsException**
Bases: Exception

The requested account already exists

**exception** beem.exceptions.**AssetDoesNotExistsException**
Bases: Exception

The asset does not exist

**exception** beem.exceptions.**BatchedCallsNotSupported**
Bases: Exception

Batch calls do not work

**exception** beem.exceptions.**BlockDoesNotExistsException**
Bases: Exception

The block does not exist

**exception** beem.exceptions.**ContentDoesNotExistsException**
Bases: Exception

The content does not exist

**exception** beem.exceptions.**InsufficientAuthorityError**
Bases: Exception

The transaction requires signature of a higher authority

**exception** beem.exceptions.**InvalidAssetException**
Bases: Exception

An invalid asset has been provided

**exception** beem.exceptions.**InvalidMessageSignature**
Bases: Exception

The message signature does not fit the message

**exception** beem.exceptions.**InvalidWifError**
Bases: Exception

The provided private Key has an invalid format

**exception** beem.exceptions.**KeyNotFound**
Bases: Exception

Key not found

**exception** beem.exceptions.**MissingKeyError**
Bases: Exception

A required key couldn't be found in the wallet

**exception** beem.exceptions.**NoWalletException**
Bases: Exception

No Wallet could be found, please use `steem.wallet.create()` to create a new wallet

**exception** beem.exceptions.**NoWriteAccess**
Bases: Exception

Cannot store to sqlite3 database due to missing write access

**exception** beem.exceptions.**ObjectNotInProposalBuffer**
Bases: Exception

Object was not found in proposal

**exception** beem.exceptions.**OfflineHasNoRPCException**
Bases: Exception

When in offline mode, we don't have RPC

**exception** beem.exceptions.**RPCConnectionRequired**
Bases: Exception

An RPC connection is required

**exception** beem.exceptions.**VestingBalanceDoesNotExistsException**
Bases: Exception

Vesting Balance does not exist

**exception** beem.exceptions.**VoteDoesNotExistsException**
Bases: Exception

The vote does not exist

**exception** beem.exceptions.**VotingInvalidOnArchivedPost**
Bases: Exception

The transaction requires signature of a higher authority

**exception** beem.exceptions.**WalletExists**
Bases: Exception

A wallet has already been created and requires a password to be unlocked by means of `steem.wallet.unlock()`.

**exception** beem.exceptions.**WalletLocked**
Bases: Exception

Wallet is locked

**exception** beem.exceptions.**WitnessDoesNotExistsException**
Bases: Exception

The witness does not exist

**exception** beem.exceptions.**WrongMasterPasswordException**
Bases: Exception

The password provided could not properly unlock the wallet

### beem.market module

**class** beem.market.**Market**(*base=None*, *quote=None*, *steem_instance=None*)
Bases: dict

This class allows to easily access Markets on the blockchain for trading, etc.

**Parameters**

- **steem_instance** (beem.steem.Steem) – Steem instance
- **base** (beem.asset.Asset) – Base asset
- **quote** (beem.asset.Asset) – Quote asset

**Returns** Blockchain Market

**Return type** dictionary with overloaded methods

Instances of this class are dictionaries that come with additional methods (see below) that allow dealing with a market and it's corresponding functions.

This class tries to identify **two** assets as provided in the parameters in one of the following forms:

- base and quote are valid assets (according to *beem.asset.Asset*)
- base:quote separated with :
- base/quote separated with /
- base-quote separated with -

---

**Note:** Throughout this library, the quote symbol will be presented first (e.g. STEEM:SBD with STEEM being the quote), while the base only refers to a secondary asset for a trade. This means, if you call *beem.market.Market.sell()* or *beem.market.Market.buy()*, you will sell/buy **only quote** and obtain/pay **only base**.

---

**accountopenorders**(*account=None*, *raw_data=False*)
Returns open Orders

**Parameters** **account** (*steem.account.Account*) – Account name or instance of Account to show orders for in this market

**buy**(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *orderid=None*, *returnOrderId=False*)
Places a buy order in a given market

**Parameters**

- **price** (*float*) – price denoted in base/quote
- **amount** (*number*) – Amount of quote to buy
- **expiration** (*number*) – (optional) expiration time of the order in seconds (defaults to 7 days)
- **killfill** (*bool*) – flag that indicates if the order shall be killed if it is not filled (defaults to False)
- **account** (*string*) – Account name that executes that order
- **returnOrderId** (*string*) – If set to "head" or "irreversible" the call will wait for the tx to appear in the head/irreversible block and add the key "orderid" to the tx output

Prices/Rates are denoted in 'base', i.e. the USD_BTS market is priced in BTS per USD.

**Example:** in the USD_BTS market, a price of 300 means a USD is worth 300 BTS

---

**Note:** All prices returned are in the **reversed** orientation as the market. I.e. in the BTC/BTS market, prices are BTS per BTC. That way you can multiply prices with *1.05* to get a +5%.

---

**Warning:** Since buy orders are placed as limit-sell orders for the base asset, you may end up obtaining more of the buy asset than you placed the order for. Example:

- You place and order to buy 10 USD for 100 BTS/USD
- This means that you actually place a sell order for 1000 BTS in order to obtain **at least** 10 USD
- If an order on the market exists that sells USD for cheaper, you will end up with more than 10 USD

**cancel**(*orderNumbers*, *account=None*, ***kwargs*)
Cancels an order you have placed in a given market. Requires only the "orderNumbers". An order number takes the form `1.7.xxx`. :param str orderNumbers: The Order Object ide of the form `1.7.xxxx`

**get_string**(*separator=':'*)
Return a formated string that identifies the market, e.g. `STEEM:SBD`

**Parameters** **separator** (*str*) – The separator of the assets (defaults to `:`)

**market_history**(*bucket_seconds=300*, *start_age=3600*, *end_age=0*)
Return the market history (filled orders). :param int bucket_seconds: Bucket size in seconds (see *returnMarketHistoryBuckets()*) :param int start_age: Age (in seconds) of the start of the window (default: 1h/3600) :param int end_age: Age (in seconds) of the end of the window (default: now/0) Example: .. code-block:: js

**{'close_sbd': 2493387,** 'close_steem': 7743431, 'high_sbd': 1943872, 'high_steem': 5999610, 'id': '7.1.5252', 'low_sbd': 534928, 'low_steem': 1661266, 'open': '2016-07-08T11:25:00', 'open_sbd': 534928, 'open_steem': 1661266, 'sbd_volume': 9714435, 'seconds': 300, 'steem_volume': 30088443},

**market_history_buckets**()

**orderbook**(*limit=25*, *raw_data=False*)
Returns the order book for SBD/STEEM market. :param int limit: Limit the amount of orders (default: 25) Sample output: .. code-block:: js

{'bids': [0.003679 USD/BTS (1.9103 USD|519.29602 BTS), 0.003676 USD/BTS (299.9997 USD|81606.16394 BTS), 0.003665 USD/BTS (288.4618 USD|78706.21881 BTS), 0.003665 USD/BTS (3.5285 USD|962.74409 BTS), 0.003665 USD/BTS (72.5474 USD|19794.41299 BTS)], 'asks': [0.003738 USD/BTS (36.4715 USD|9756.17339 BTS), 0.003738 USD/BTS (18.6915 USD|5000.00000 BTS), 0.003742 USD/BTS (182.6881 USD|48820.22081 BTS), 0.003772 USD/BTS (4.5200 USD|1198.14798 BTS), 0.003799 USD/BTS (148.4975 USD|39086.59741 BTS)]}

---

**Note:** Each bid is an instance of class:*beem.price.Order* and thus carries the keys `base`, `quote` and `price`. From those you can obtain the actual amounts for sale

---

**recent_trades**(*limit=25*, *raw_data=False*)
Returns the order book for a given market. You may also specify "all" to get the orderbooks of all markets.

**Parameters** **limit** (*int*) – Limit the amount of orders (default: 25)

Sample output:

```
{'bids': [0.003679 USD/BTS (1.9103 USD|519.29602 BTS),
0.003676 USD/BTS (299.9997 USD|81606.16394 BTS),
0.003665 USD/BTS (288.4618 USD|78706.21881 BTS),
0.003665 USD/BTS (3.5285 USD|962.74409 BTS),
0.003665 USD/BTS (72.5474 USD|19794.41299 BTS)],
'asks': [0.003738 USD/BTS (36.4715 USD|9756.17339 BTS),
0.003738 USD/BTS (18.6915 USD|5000.00000 BTS),
0.003742 USD/BTS (182.6881 USD|48820.22081 BTS),
0.003772 USD/BTS (4.5200 USD|1198.14798 BTS),
0.003799 USD/BTS (148.4975 USD|39086.59741 BTS)]}
```

---

**Note:** Each bid is an instance of class:*steem.price.Order* and thus carries the keys `base`, `quote` and

price. From those you can obtain the actual amounts for sale

**sell**(*price*, *amount*, *expiration=None*, *killfill=False*, *account=None*, *orderid=None*, *returnOrderId=False*)
Places a sell order in a given market

**Parameters**

- **price** (*float*) – price denoted in base/quote
- **amount** (*number*) – Amount of quote to sell
- **expiration** (*number*) – (optional) expiration time of the order in seconds (defaults to 7 days)
- **killfill** (*bool*) – flag that indicates if the order shall be killed if it is not filled (defaults to False)
- **account** (*string*) – Account name that executes that order
- **returnOrderId** (*string*) – If set to "head" or "irreversible" the call will wait for the tx to appear in the head/irreversible block and add the key "orderid" to the tx output

Prices/Rates are denoted in 'base', i.e. the USD_BTS market is priced in BTS per USD.

**Example:** in the USD_BTS market, a price of 300 means a USD is worth 300 BTS

**Note:** All prices returned are in the **reversed** orientation as the market. I.e. in the BTC/BTS market, prices are BTS per BTC. That way you can multiply prices with *1.05* to get a +5%.

**ticker**(*raw_data=False*)
Returns the ticker for all markets.

Output Parameters:

- latest: Price of the order last filled
- lowest_ask: Price of the lowest ask
- highest_bid: Price of the highest bid
- sbd_volume: Volume of SBD
- steem_volume: Volume of STEEM
- percent_change: 24h change percentage (in %)

**Note:** Market is STEEM:SBD and prices are SBD per STEEM!

Sample Output:

```
{'highest_bid': 0.30100226633322913,
 'latest': 0.0,
 'lowest_ask': 0.3249636958897082,
 'percent_change': 0.0,
 'sbd_volume': 108329611.0,
 'steem_volume': 355094043.0}
```

**trade_history**(*start=None*, *stop=None*, *intervall=None*, *limit=25*, *raw_data=False*)
Returns the trade history for the internal market

This function allows to fetch a fixed number of trades at fixed intervall times to reduce the call duration time. E.g. it is possible to receive the trades from the last 7 days, by fetching 100 trades each 6 hours.

When intervall is set to None, all trades are recieved between start and stop. This can take a while.

**Parameters**

- **start** (*datetime*) – Start date
- **stop** (*datetime*) – Stop date
- **intervall** (*timedelta*) – Defines the intervall
- **limit** (*int*) – Defines how many trades are fetched at each intervall point
- **raw_data** (*bool*) – when True, the raw data are returned

**trades**(*limit=100*, *start=None*, *stop=None*, *raw_data=False*)

Returns your trade history for a given market.

**Parameters**

- **limit** (*int*) – Limit the amount of orders (default: 100)
- **start** (*datetime*) – start time
- **stop** (*datetime*) – stop time

**volume24h**(*raw_data=False*)

Returns the 24-hour volume for all markets, plus totals for primary currencies.

Sample output:

```
{
    "STEEM": 361666.63617,
    "SBD": 1087.0
}
```

## beem.memo module

**class** beem.memo.**Memo**(*from_account=None*, *to_account=None*, *steem_instance=None*)

Bases: object

Deals with Memos that are attached to a transfer

**Parameters**

- **from_account** (beem.account.Account) – Account that has sent the memo
- **to_account** (beem.account.Account) – Account that has received the memo
- **steem_instance** (beem.steem.Steem) – Steem instance

A memo is encrypted with a shared secret derived from a private key of the sender and a public key of the receiver. Due to the underlying mathematics, the same shared secret can be derived by the private key of the receiver and the public key of the sender. The encrypted message is perturbed by a nonce that is part of the transmitted message.

```
from beem.memo import Memo
m = Memo("steemeu", "wallet.xeroc")
m.steem.wallet.unlock("secret")
enc = (m.encrypt("foobar"))
print(enc)
```

(continues on next page)

(continued from previous page)

```
>> {'nonce': '17329630356955254641', 'message': '8563e2bb2976e0217806d642901a2855
↪'}
print(m.decrypt(enc))
>> foobar
```

To decrypt a memo, simply use

```
from beem.memo import Memo
m = Memo()
m.steem.wallet.unlock("secret")
print(m.decrypt(op_data["memo"]))
```

if `op_data` being the payload of a transfer operation.

In Steem, memos are AES-256 encrypted with a shared secret between sender and receiver. It is derived from the memo private key of the sender and the memo publick key of the receiver.

In order for the receiver to decode the memo, the shared secret has to be derived from the receiver's private key and the senders public key.

The memo public key is part of the account and can be retreived with the *get_account* call:

```
get_account <accountname>
{
  [...]
  "options": {
    "memo_key": "GPH5TPTziKkLexhVKsQKtSpo4bAv5RnB8oXcG4sMHEwCcTf3r7dqE",
    [...]
  },
  [...]
}
```

while the memo private key can be dumped with *dump_private_keys*

The take the following form:

```
{
  "from": "GPH5mgup8evDqMnT86L7scVebRYDC2fwAWmygPEUL43LjstQegYCC",
  "to": "GPH5Ar4j53kFWuEZQ9XhxbAja4YXMPJ2EnUg5QcrdeMFYUNMMNJbe",
  "nonce": "13043867485137706821",
  "message": "d55524c37320920844ca83bb20c8d008"
}
```

The fields *from* and *to* contain the memo public key of sender and receiver. The *nonce* is a random integer that is used for the seed of the AES encryption of the message.

The high level memo class makes use of the pysteem wallet to obtain keys for the corresponding accounts.

```
from beem.memo import Memo
from beem.account import Account

memoObj = Memo(
    from_account=Account(from_account),
    to_account=Account(to_account)
)
encrypted_memo = memoObj.encrypt(memo)
```

```
from getpass import getpass
from beem.block import Block
from beem.memo import Memo

# Obtain a transfer from the blockchain
block = Block(23755086)                     # block
transaction = block["transactions"][3]      # transactions
op = transaction["operations"][0]           # operation
op_id = op[0]                               # operation type
op_data = op[1]                             # operation payload

# Instantiate Memo for decoding
memo = Memo()

# Unlock wallet
memo.unlock_wallet(getpass())

# Decode memo
# Raises exception if required keys not available in the wallet
print(memo.decrypt(op_data["transfer"]))
```

**decrypt**(*memo*)
Decrypt a memo

**Parameters** **memo** (*str*) – encrypted memo message

**Returns** encrypted memo

**Return type** str

**encrypt**(*memo*, *bts_encrypt=False*)
Encrypt a memo

**Parameters** **memo** (*str*) – clear text memo message

**Returns** encrypted memo

**Return type** str

**unlock_wallet**(**args*, ***kwargs*)
Unlock the library internal wallet

### beem.message module

**class** beem.message.**Message**(*message*, *steem_instance=None*)
Bases: object

**sign**(*account=None*, ***kwargs*)
Sign a message with an account's memo key

**Parameters** **account** (*str*) – (optional) the account that owns the bet (defaults to default_account)

**Returns** the signed message encapsulated in a known format

**verify**(***kwargs*)
Verify a message with an account's memo key

**Parameters** **account** (*str*) – (optional) the account that owns the bet (defaults to default_account)

**Returns** True if the message is verified successfully

:raises InvalidMessageSignature if the signature is not ok

### beem.notify module

**class** beem.notify.**Notify**(*on_block=None*, *only_block_id=False*, *steem_instance=None*, *keep_alive=25*)

Bases: events.events.Events

Notifications on Blockchain events.

This modules allows yout to be notified of events taking place on the blockchain.

**Parameters**

- **on_block** (*fnt*) – Callback that will be called for each block received
- **steem_instance** (beem.steem.Steem) – Steem instance

**Example**

```
from pprint import pprint
from beem.notify import Notify

notify = Notify(
    on_block=print,
)
notify.listen()
```

**close**()

Cleanly close the Notify instance

**listen**()

This call initiates the listening/notification process. It behaves similar to run_forever().

**process_block**(*message*)

**reset_subscriptions**(*accounts=[]*)

Change the subscriptions of a running Notify instance

### beem.price module

**class** beem.price.**FilledOrder**(*order*, *steem_instance=None*, ***kwargs*)

Bases: *beem.price.Price*

This class inherits *beem.price.Price* but has the base and quote Amounts not only be used to represent the price (as a ratio of base and quote) but instead has those amounts represent the amounts of an actually filled order!

**Parameters** **steem_instance** (beem.steem.Steem) – Steem instance

---

**Note:** Instances of this class come with an additional date key that shows when the order has been filled!

---

**json**()

**class** beem.price.**Order**(*base*, *quote=None*, *steem_instance=None*, ***kwargs*)

Bases: *beem.price.Price*

This class inherits *beem.price.Price* but has the base and quote Amounts not only be used to represent the price (as a ratio of base and quote) but instead has those amounts represent the amounts of an actual order!

**Parameters** **steem_instance** (beem.steem.Steem) – Steem instance

---

**Note:** If an order is marked as deleted, it will carry the 'deleted' key which is set to True and all other data be None.

---

**class** beem.price.**Price**(*price=None*, *base=None*, *quote=None*, *base_asset=None*, *steem_instance=None*)

Bases: dict

This class deals with all sorts of prices of any pair of assets to simplify dealing with the tuple:

```
(quote, base)
```

each being an instance of *beem.amount.Amount*. The amount themselves define the price.

---

**Note:** The price (floating) is derived as base/quote

---

**Parameters**

- **args** (*list*) – Allows to deal with different representations of a price
- **base** (beem.asset.Asset) – Base asset
- **quote** (beem.asset.Asset) – Quote asset
- **steem_instance** (beem.steem.Steem) – Steem instance

**Returns** All data required to represent a price

**Return type** dict

Way to obtain a proper instance:

- args is a str with a price and two assets
- args can be a floating number and base and quote being instances of *beem.asset.Asset*
- args can be a floating number and base and quote being instances of str
- args can be dict with keys price, base, and quote (*graphene balances*)
- args can be dict with keys base and quote
- args can be dict with key receives (filled orders)
- args being a list of [quote, base] both being instances of *beem.amount.Amount*
- args being a list of [quote, base] both being instances of str (amount symbol)
- base and quote being instances of beem.asset.Amount

This allows instanciations like:

- Price("0.315 SBD/STEEM")
- Price(0.315, base="SBD", quote="STEEM")
- Price(0.315, base=Asset("SBD"), quote=Asset("STEEM"))

- `Price({"base": {"amount": 1, "asset_id": "SBD"}, "quote": {"amount": 10, "asset_id": "SBD"}})`
- `Price(quote="10 STEEM", base="1 SBD")`
- `Price("10 STEEM", "1 SBD")`
- `Price(Amount("10 STEEM"), Amount("1 SBD"))`
- `Price(1.0, "SBD/STEEM")`

Instances of this class can be used in regular mathematical expressions (`+-*/%`) such as:

```
>>> from beem.price import Price
>>> Price("0.3314 SBD/STEEM") * 2
0.662600000 SBD/STEEM
```

**as_base**(*base*)
Returns the price instance so that the base asset is `base`.

Note: This makes a copy of the object!

**as_quote**(*quote*)
Returns the price instance so that the quote asset is `quote`.

Note: This makes a copy of the object!

**copy**() → a shallow copy of D

**invert**()
Invert the price (e.g. go from `SBD/STEEM` into `STEEM/SBD`)

**json**()

**market**
Open the corresponding market

**Returns** Instance of *beem.market.Market* for the corresponding pair of assets.

**symbols**()

### beem.storage module

**class** beem.storage.**Configuration**
Bases: *beem.storage.DataDir*

This is the configuration storage that stores key/value pairs in the *config* table of the SQLite3 database.

**checkBackup**()
Backup the SQL database every 7 days

**config_defaults = {'node': ['wss://steemd.privex.io', 'wss://steemd.pevo.science', 'w**

**create_table**()
Create the new table in the SQLite database

**delete**(*key*)
Delete a key from the configuration store

**exists_table**()
Check if the database table exists

**get**(*key*, *default=None*)
Return the key if exists or a default value

**items**()

**nodes = ['wss://steemd.privex.io', 'wss://steemd.pevo.science', 'wss://rpc.buildteam.i**
Default configuration

**class** beem.storage.**DataDir**
Bases: object

This class ensures that the user's data is stored in its OS preotected user directory:

**OSX:**

- *~/Library/Application Support/<AppName>*

**Windows:**

- *C:Documents and Settings<User>Application DataLocal Settings<AppAuthor><AppName>*
- *C:Documents and Settings<User>Application Data<AppAuthor><AppName>*

**Linux:**

- *~/.local/share/<AppName>*

Furthermore, it offers an interface to generated backups in the *backups/* directory every now and then.

**appauthor = 'beem'**

**appname = 'beem'**

**clean_data**()
Delete files older than 70 days

**data_dir = '/home/docs/.local/share/beem'**

**mkdir_p**()
Ensure that the directory in which the data is stored exists

**recover_with_latest_backup**(*backupdir='backups'*)
Replace database with latest backup

**refreshBackup**()
Make a new backup

**sqlDataBaseFile = '/home/docs/.local/share/beem/beem.sqlite'**

**sqlite3_backup**(*backupdir*)
Create timestamped database copy

**sqlite3_copy**(*src*, *dst*)
Copy sql file from src to dst

**storageDatabase = 'beem.sqlite'**

**class** beem.storage.**Key**
Bases: *beem.storage.DataDir*

This is the key storage that stores the public key and the (possibly encrypted) private key in the *keys* table in the SQLite3 database.

**add**(*wif*, *pub*)

**Add a new public/private key pair (correspondence has to be** checked elsewhere!)

**Parameters**

- **pub** (*str*) – Public key
- **wif** (*str*) – Private key

**create_table**()
Create the new table in the SQLite database

**delete**(*pub*)
Delete the key identified as *pub*

**Parameters** **pub** (*str*) – Public key

**exists_table**()
Check if the database table exists

**getPrivateKeyForPublicKey**(*pub*)

**Returns the (possibly encrypted) private key that** corresponds to a public key

**Parameters** **pub** (*str*) – Public key

The encryption scheme is BIP38

**getPublicKeys**()
Returns the public keys stored in the database

**updateWif**(*pub*, *wif*)
Change the wif to a pubkey

**Parameters**

- **pub** (*str*) – Public key
- **wif** (*str*) – Private key

**wipe**(*sure=False*)
Purge the entire wallet. No keys will survive this!

**class** beem.storage.**MasterPassword**(*password*)
Bases: object

The keys are encrypted with a Masterpassword that is stored in the configurationStore. It has a checksum to verify correctness of the password

**changePassword**(*newpassword*)
Change the password

**config_key = 'encrypted_master_password'**
This key identifies the encrypted master password stored in the confiration

**decryptEncryptedMaster**()
Decrypt the encrypted masterpassword

**decrypted_master = ''**

**deriveChecksum**(*s*)
Derive the checksum

**getEncryptedMaster**()
Obtain the encrypted masterkey

**newMaster**()
Generate a new random masterpassword

**password = ''**

**saveEncrytpedMaster**()
Store the encrypted master password in the configuration store

**static wipe**(*sure=False*)
Remove all keys from configStorage

### beem.transactionbuilder module

**class** beem.transactionbuilder.**TransactionBuilder**(*tx={}*, *expiration=None*, *steem_instance=None*)

Bases: dict

This class simplifies the creation of transactions by adding operations and signers. To build your own transactions and sign them

param dict tx: transaction (Optional). If not set, the new transaction is created. param str expiration: expiration date param Steem steem_instance: If not set, shared_steem_instance() is used

```
from beem.transactionbuilder import TransactionBuilder
from beembase.operations import Transfer
tx = TransactionBuilder()
tx.appendOps(Transfer(**{
          "from": "test",
          "to": "test1",
          "amount": "1 STEEM",
          "memo": ""
      }))
tx.appendSigner("test", "active")
tx.sign()
tx.broadcast()
```

**addSigningInformation**(*account*, *permission*, *reconstruct_tx=False*)
This is a private method that adds side information to a unsigned/partial transaction in order to simplify later signing (e.g. for multisig or coldstorage)

Not needed when "appendWif" was already or is going to be used

FIXME: Does not work with owner keys!

**Parameters** **reconstruct_tx** (*bool*) – when set to False and tx is already contructed, it will not reconstructed and already added signatures remain

**appendMissingSignatures**()
Store which accounts/keys are supposed to sign the transaction

This method is used for an offline-signer!

**appendOps**(*ops*, *append_to=None*)
Append op(s) to the transaction builder

**Parameters** **ops** (*list*) – One or a list of operations

**appendSigner**(*account*, *permission*)
Try to obtain the wif key from the wallet by telling which account and permission is supposed to sign the transaction It is possible to add more than one signer.

**appendWif**(*wif*)
Add a wif that should be used for signing of the transaction.

**Parameters** **wif** (*string*) – One wif key to use for signing a transaction.

**broadcast**(*max_block_age=-1*)
Broadcast a transaction to the steem network Returns the signed transaction and clears itself after broadast

Clears itself when broadcast was not sucessfully.

**Parameters** **max_block_age** (*int*) – paramerter only used for appbase ready nodes

**clear**()
Clear the transaction builder and start from scratch

**clearWifs**()
Clear all stored wifs

**constructTx**()
Construct the actual transaction and store it in the class's dict store

**get_parent**()
TransactionBuilders don't have parents, they are their own parent

**is_empty**()
Check if ops is empty

**json**()
Show the transaction as plain json

**list_operations**()
List all ops

**set_expiration**(*p*)
Set expiration date

**sign**(*reconstruct_tx=True*)
Sign a provided transaction with the provided key(s) One or many wif keys to use for signing a transaction. The wif keys can be provided by "appendWif" or the signer can be defined "appendSigner". The wif keys from all signer that are defined by "appendSigner will be loaded from the wallet.

**Parameters** **reconstruct_tx** (*bool*) – when set to False and tx is already contructed, it will not reconstructed and already added signatures remain

**verify_authority**()
Verify the authority of the signed transaction

### beem.utils module

beem.utils.**addTzInfo**(*t*, *timezone='UTC'*)
Returns a datetime object with tzinfo added

beem.utils.**assets_from_string**(*text*)
Correctly split a string containing an asset pair.

Splits the string into two assets with the separator being on of the following: `:`, `/`, or `-`.

beem.utils.**construct_authorperm**(**args*)
Create a post identifier from comment/post object or arguments. Examples:

beem.utils.**construct_authorpermvoter**(**args*)
Create a vote identifier from vote object or arguments. Examples:

beem.utils.**derive_permlink**(*title*, *parent_permlink=None*, *parent_author=None*)

beem.utils.**findall_patch_hunks**(*body=None*)

beem.utils.**formatTime**(*t*)
Properly Format Time for permlinks

beem.utils.**formatTimeFromNow**(*secs=0*)
Properly Format Time that is *x* seconds in the future

**Parameters** **secs** (*int*) – Seconds to go in the future (*x>0*) or the past (*x<0*)

**Returns** Properly formated time for Graphene (*%Y-%m-%dT%H:%M:%S*)

**Return type** str

beem.utils.**formatTimeString**(*t*)
Properly Format Time for permlinks

beem.utils.**formatTimedelta**(*td*)
Format timedelta to String

beem.utils.**get_node_list**(*appbase=False*, *testing=False*)
Returns node list

beem.utils.**make_patch**(*a*, *b*, *n=3*)

beem.utils.**parse_time**(*block_time*)
Take a string representation of time from the blockchain, and parse it into datetime object.

beem.utils.**remove_from_dict**(*obj*, *keys=[]*, *keep_keys=True*)
Prune a class or dictionary of all but keys (keep_keys=True). Prune a class or dictionary of specified keys.(keep_keys=False).

beem.utils.**reputation_to_score**(*rep*)
Converts the account reputation value into the reputation score

beem.utils.**resolve_authorperm**(*identifier*)
Correctly split a string containing an authorperm.

Splits the string into author and permlink with the following separator: `/`.

beem.utils.**resolve_authorpermvoter**(*identifier*)
Correctly split a string containing an authorpermvoter.

Splits the string into author and permlink with the following separator: `/` and `|`.

beem.utils.**resolve_root_identifier**(*url*)

beem.utils.**sanitize_permlink**(*permlink*)

### beem.vote module

**class** beem.vote.**AccountVotes**(*account*, *start=None*, *stop=None*, *steem_instance=None*)
Bases: *beem.vote.VotesObject*

Obtain a list of votes for an account Lists the last 100+ votes on the given account.

**Parameters**

- **account** (*str*) – Account name
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.vote.**ActiveVotes**(*authorperm*, *steem_instance=None*)
Bases: *beem.vote.VotesObject*

Obtain a list of votes for a post

**Parameters**

- **authorperm** (*str*) – authorperm link
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.vote.**Vote**(*voter*, *authorperm=None*, *full=False*, *lazy=False*, *steem_instance=None*)
Bases: beem.blockchainobject.BlockchainObject

Read data about a Vote in the chain

**Parameters**

- **authorperm** (*str*) – perm link to post/comment
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

```
from beem.vote import Vote
v = Vote("theaussiegame/cryptokittie-giveaway-number-2|")
```

**json**()

**percent**

**refresh**()

**rep**

**reputation**

**rshares**

**sbd**

**time**

**type_id = 11**

**votee**

**voter**

**weight**

**class** beem.vote.**VotesObject**
Bases: list

**get_list**(*var='voter'*, *voter=None*, *votee=None*, *start=None*, *stop=None*, *start_percent=None*, *stop_percent=None*, *sort_key='time'*, *reverse=True*)

**get_sorted_list**(*sort_key='time'*, *reverse=True*)

**printAsTable**(*voter=None*, *votee=None*, *start=None*, *stop=None*, *start_percent=None*, *stop_percent=None*, *sort_key='time'*, *reverse=True*, *allow_refresh=True*, *return_str=False*, ***kwargs*)

**print_stats**(*return_str=False*, ***kwargs*)

### beem.wallet module

**class** beem.wallet.**Wallet**(*steem_instance=None*, **args*, ***kwargs*)

Bases: object

The wallet is meant to maintain access to private keys for your accounts. It either uses manually provided private keys or uses a SQLite database managed by storage.py.

**Parameters**

- **rpc** (SteemNodeRPC) – RPC connection to a Steem node
- **keys** (*array,dict,string*) – Predefine the wif keys to shortcut the wallet database

Three wallet operation modes are possible:

- **Wallet Database**: Here, beem loads the keys from the locally stored wallet SQLite database (see storage.py). To use this mode, simply call Steem() without the keys parameter
- **Providing Keys**: Here, you can provide the keys for your accounts manually. All you need to do is add the wif keys for the accounts you want to use as a simple array using the keys parameter to Steem().
- **Force keys**: This more is for advanced users and requires that you know what you are doing. Here, the keys parameter is a dictionary that overwrite the active, owner, posting or memo keys for any account. This mode is only used for *foreign* signatures!

A new wallet can be created by using:

```
from beem import Steem
steem = Steem()
steem.wallet.wipe(True)
steem.wallet.create("supersecret-passphrase")
```

This will raise an exception if you already have a wallet installed.

The wallet can be unlocked for signing using

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("supersecret-passphrase")
```

A private key can be added by using the steem.wallet.Wallet.addPrivateKey() method that is available **after** unlocking the wallet with the correct passphrase:

```
from beem import Steem
steem = Steem()
steem.wallet.unlock("supersecret-passphrase")
steem.wallet.addPrivateKey("5xxxxxxxxxxxxxxxxxxx")
```

---

**Note:** The private key has to be either in hexadecimal or in wallet import format (wif) (starting with a 5).

---

**MasterPassword = None**

**addPrivateKey**(*wif*)

Add a private key to the wallet database

**changePassphrase**(*new_pwd*)

Change the passphrase for the wallet database

**clear_local_keys**()
Clear all manually provided keys

**configStorage = None**

**create**(*pwd*)
Alias for newWallet()

**created**()
Do we have a wallet database already?

**decrypt_wif**(*encwif*)
decrypt a wif key

**encrypt_wif**(*wif*)
Encrypt a wif key

**getAccount**(*pub*)
Get the account data for a public key (first account found for this public key)

**getAccountFromPrivateKey**(*wif*)
Obtain account name from private key

**getAccountFromPublicKey**(*pub*)
Obtain the first account name from public key

**getAccounts**()
Return all accounts installed in the wallet database

**getAccountsFromPublicKey**(*pub*)
Obtain all accounts associated with a public key

**getActiveKeyForAccount**(*name*)
Obtain owner Active Key for an account from the wallet database

**getAllAccounts**(*pub*)
Get the account data for a public key (all accounts found for this public key)

**getKeyForAccount**(*name*, *key_type*)
Obtain *key_type* Private Key for an account from the wallet database

**getKeyType**(*account*, *pub*)
Get key type

**getMemoKeyForAccount**(*name*)
Obtain owner Memo Key for an account from the wallet database

**getOwnerKeyForAccount**(*name*)
Obtain owner Private Key for an account from the wallet database

**getPostingKeyForAccount**(*name*)
Obtain owner Posting Key for an account from the wallet database

**getPrivateKeyForPublicKey**(*pub*)
Obtain the private key for a given public key

**Parameters** **pub** (*str*) – Public Key

**getPublicKeys**()
Return all installed public keys

**keyMap = {}**

**keyStorage = None**

**keys = {}**

**lock**()
Lock the wallet database

**locked**()
Is the wallet database locked?

**masterpassword = None**

**newWallet**(*pwd*)
Create a new wallet database

**prefix**

**removeAccount**(*account*)
Remove all keys associated with a given account

**removePrivateKeyFromPublicKey**(*pub*)
Remove a key from the wallet database

**rpc**

**setKeys**(*loadkeys*)
This method is strictly only for in memory keys that are passed to Wallet/Steem with the `keys` argument

**tryUnlockFromEnv**()
Try to fetch the unlock password from UNLOCK environment variable and keyring when no password is given.

**unlock**(*pwd=None*)
Unlock the wallet database

**unlocked**()
Is the wallet database unlocked?

**wipe**(*sure=False*)
Purge all data in wallet database

### beem.witness module

**class** beem.witness.**ListWitnesses**(*from_account*, *limit*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses which have been voted by an account

**Parameters**

- **from_account** (*str*) – Account name
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.witness.**Witness**(*owner*, *full=False*, *lazy=False*, *steem_instance=None*)
Bases: beem.blockchainobject.BlockchainObject

Read data about a witness in the chain

**Parameters**

- **account_name** (*str*) – Name of the witness
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

```
from beem.witness import Witness
Witness("gtg")
```

**account**

**feed_publish**(*base*, *quote='1.000 STEEM'*, *account=None*)
Publish a feed price as a witness. :param float base: USD Price of STEEM in SBD (implied price) :param float quote: (optional) Quote Price. Should be 1.000, unless we are adjusting the feed to support the peg. :param str account: (optional) the source account for the transfer if not self["owner"]

**is_active**

**refresh**()

**type_id = 3**

**update**(*signing_key*, *url*, *props*, *account=None*)
Update witness :param pubkey signing_key: Signing key :param str url: URL :param dict props: Properties :param str account: (optional) witness account name

**Properties:::**

{ "account_creation_fee": x, "maximum_block_size": x, "sbd_interest_rate": x,

}

**class** beem.witness.**Witnesses**(*steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of **active** witnesses and the current schedule

**Parameters** **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.witness.**WitnessesObject**
Bases: list

**printAsTable**(*sort_key='votes'*, *reverse=True*, *return_str=False*, ***kwargs*)

**class** beem.witness.**WitnessesRankedByVote**(*from_account=''*, *limit=100*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses ranked by Vote

**Parameters**

- **from_account** (*str*) – Witness name
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

**class** beem.witness.**WitnessesVotedByAccount**(*account*, *steem_instance=None*)
Bases: *beem.witness.WitnessesObject*

Obtain a list of witnesses which have been voted by an account

**Parameters**

- **account** (*str*) – Account name
- **steem_instance** (*steem*) – Steem() instance to use when accesing a RPC

## Module contents

beem.

## 4.2 beembase

### 4.2.1 beembase package

#### Submodules

#### beembase.chains module

#### beembase.memo module

beembase.memo.**decode_memo**(*priv*, *message*)
Decode a message with a shared secret between Alice and Bob :param PrivateKey priv: Private Key (of Bob) :param base58encoded message: Encrypted Memo message :return: Decrypted message :rtype: str :raise ValueError: if message cannot be decoded as valid UTF-8

string

beembase.memo.**decode_memo_bts**(*priv*, *pub*, *nonce*, *message*)
Decode a message with a shared secret between Alice and Bob

**Parameters**

- **priv** (PrivateKey) – Private Key (of Bob)
- **pub** (PublicKey) – Public Key (of Alice)
- **nonce** (*int*) – Nonce used for Encryption
- **message** (*bytes*) – Encrypted Memo message

**Returns** Decrypted message

**Return type** str

**Raises** **ValueError** – if message cannot be decoded as valid UTF-8 string

beembase.memo.**encode_memo**(*priv*, *pub*, *nonce*, *message*, ***kwargs*)
Encode a message with a shared secret between Alice and Bob :param PrivateKey priv: Private Key (of Alice) :param PublicKey pub: Public Key (of Bob) :param int nonce: Random nonce :param str message: Memo message :return: Encrypted message :rtype: hex

beembase.memo.**encode_memo_bts**(*priv*, *pub*, *nonce*, *message*)
Encode a message with a shared secret between Alice and Bob

**Parameters**

- **priv** (PrivateKey) – Private Key (of Alice)
- **pub** (PublicKey) – Public Key (of Bob)
- **nonce** (*int*) – Random nonce
- **message** (*str*) – Memo message

**Returns** Encrypted message

**Return type** hex

beembase.memo.**get_shared_secret**(*priv*, *pub*)
Derive the share secret between `priv` and `pub`

**Parameters**

- **priv** (Base58) – Private Key
- **pub** (Base58) – Public Key

**Returns** Shared secret

**Return type** hex

The shared secret is generated such that:

```
Pub(Alice) * Priv(Bob) = Pub(Bob) * Priv(Alice)
```

beembase.memo.**init_aes**(*shared_secret*, *nonce*)
Initialize AES instance :param hex shared_secret: Shared Secret to use as encryption key :param int nonce: Random nonce :return: AES instance and checksum of the encryption key :rtype: length 2 tuple

beembase.memo.**init_aes_bts**(*shared_secret*, *nonce*)
Initialize AES instance

**Parameters**

- **shared_secret** (*hex*) – Shared Secret to use as encryption key
- **nonce** (*int*) – Random nonce

**Returns** AES instance

**Return type** AES

### beembase.objects module

**class** beembase.objects.**Amount**(*d*)
Bases: object

**class** beembase.objects.**Beneficiaries**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

**class** beembase.objects.**Beneficiary**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

**class** beembase.objects.**CommentOptionExtensions**(*o*)
Bases: beemgraphenebase.types.Static_variant

Serialize Comment Payout Beneficiaries. Args:

beneficiaries (list): A static_variant containing beneficiaries.

**Example:**

**::**

**[0,**

**{'beneficiaries': [** {'account': 'furion', 'weight': 10000}

]}

]

**class** beembase.objects.**ExchangeRate**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

**class** beembase.objects.**Extension**(*d*)
Bases: beemgraphenebase.types.Array

**class** beembase.objects.**Memo**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

**class** beembase.objects.**Operation**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.Operation*

**getOperationNameForId**(*i*)
Convert an operation id into the corresponding string

**json**()

**operations**()

**class** beembase.objects.**Permission**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

**class** beembase.objects.**Price**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

**class** beembase.objects.**WitnessProps**(**args*, ***kwargs*)
Bases: *beemgraphenebase.objects.GrapheneObject*

### beembase.objecttypes module

beembase.objecttypes.**object_type = {'account': 2, 'account_history': 18, 'block_summary'**
Object types for object ids

### beembase.operationids module

beembase.operationids.**getOperationNameForId**(*i*)
Convert an operation id into the corresponding string

beembase.operationids.**ops = ['vote', 'comment', 'transfer', 'transfer_to_vesting', 'withdra**
Operation ids

### beembase.operations module

beembase.operationids.**getOperationNameForId**(*i*)
Convert an operation id into the corresponding string

beembase.operationids.**ops = ['vote', 'comment', 'transfer', 'transfer_to_vesting', 'withdra**
Operation ids

### beembase.transactions module

beembase.transactions.**getBlockParams**(*ws*)
Auxiliary method to obtain `ref_block_num` and `ref_block_prefix`. Requires a websocket connection to a witness node!

### Module contents

beembase.

## 4.3 beemapi

### 4.3.1 beemapi package

#### Submodules

#### SteemNodeRPC

This class allows to call API methods exposed by the witness node via websockets.

#### Defintion

**class** beemapi.steemnoderpc.**SteemNodeRPC**(**args*, ***kwargs*)

**This class allows to call API methods exposed by the witness node via** websockets / rpc-json.

**Parameters**

- **urls** (*str*) – Either a single Websocket/Http URL, or a list of URLs
- **user** (*str*) – Username for Authentication
- **password** (*str*) – Password for Authentication
- **num_retries** (*int*) – Try x times to num_retries to a node on disconnect, -1 for indefinitely
- **num_retries_call** (*int*) – Repeat num_retries_call times a rpc call on node error (default is 5)
- **timeout** (*int*) – Timeout setting for https nodes (default is 60)

**__getattr__**(*name*)
Map all methods to RPC calls and pass through the arguments.

**rpcexec**(*payload*)
Execute a call by sending the payload. It makes use of the GrapheneRPC library. In here, we mostly deal with Steem specific error handling

**Parameters** **payload** (*json*) – Payload data

**Raises**

- **ValueError** – if the server does not respond in proper JSON format
- **RPCError** – if the server returns an error

#### beemapi.exceptions module

**exception** beemapi.exceptions.**ApiNotSupported**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**InvalidEndpointUrl**
Bases: Exception

**exception** beemapi.exceptions.**MissingRequiredActiveAuthority**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**NoAccessApi**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**NoApiWithName**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**NoMethodWithName**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**NumRetriesReached**
Bases: Exception

**exception** beemapi.exceptions.**UnhandledRPCError**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**UnkownKey**
Bases: beemgrapheneapi.exceptions.RPCError

**exception** beemapi.exceptions.**UnnecessarySignatureDetected**
Bases: Exception

beemapi.exceptions.**decodeRPCErrorMsg**(*e*)
Helper function to decode the raised Exception and give it a python Exception class

### SteemWebsocket

This class allows subscribe to push notifications from the Steem node.

```
from pprint import pprint
from beemapi.websocket import SteemWebsocket

ws = SteemWebsocket(
    "wss://gtg.steem.house:8090",
    accounts=["test"],
    on_block=print,
)

ws.run_forever()
```

### Defintion

**class** beemapi.websocket.**SteemWebsocket**(*urls*, *user=''*, *password=''*, *only_block_id=False*, *on_block=None*, *keep_alive=25*, *num_retries=-1*, *timeout=60*, **args*, ***kwargs*)
Create a websocket connection and request push notifications

**Parameters**

- **urls** (*str*) – Either a single Websocket URL, or a list of URLs
- **user** (*str*) – Username for Authentication
- **password** (*str*) – Password for Authentication
- **keep_alive** (*int*) – seconds between a ping to the backend (defaults to 25seconds)

After instanciating this class, you can add event slots for:

- on_block

which will be called accordingly with the notification message received from the Steem node:

```
ws = SteemWebsocket(
    "wss://gtg.steem.house:8090",
)
ws.on_block += print
ws.run_forever()
```

Notices:

- `on_block`:

  ```
  '0062f19df70ecf3a478a84b4607d9ad8b3e3b607'
  ```

**_SteemWebsocket__set_subscriptions**()
set subscriptions ot on_block function

**__events__ = ['on_block']**

**__getattr__**(*name*)
Map all methods to RPC calls and pass through the arguments

**__init__**(*urls*, *user=''*, *password=''*, *only_block_id=False*, *on_block=None*, *keep_alive=25*, *num_retries=-1*, *timeout=60*, **args*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

**__module__ = 'beemapi.websocket'**

**_ping**()
Send keep_alive request

**cancel_subscriptions**()
cancel_all_subscriptions removed from api

**close**()
Closes the websocket connection and waits for the ping thread to close

**get_request_id**()
Generates next request id

**on_close**(*ws*)
Called when websocket connection is closed

**on_error**(*ws*, *error*)
Called on websocket errors

**on_message**(*ws*, *reply*, **args*)
This method is called by the websocket connection on every message that is received. If we receive a `notice`, we hand over post-processing and signalling of events to `process_notice`.

**on_open**(*ws*)
This method will be called once the websocket connection is established. It will

- login,
- register to the database api, and
- subscribe to the objects defined if there is a callback/slot available for callbacks

**process_block**(*data*)
This method is called on notices that need processing. Here, we call the `on_block` slot.

**reset_subscriptions**(*accounts=[]*)
Reset subscriptions

**rpcexec**(*payload*)
Execute a call by sending the payload.

**Parameters** **payload** (*json*) – Payload data

**Raises**

- **ValueError** – if the server does not respond in proper JSON format
- **RPCError** – if the server returns an error

**run_forever**()
This method is used to run the websocket app continuously. It will execute callbacks as defined and try to stay connected with the provided APIs

**stop**()
Stop running Websocket

### Module contents

beemapi.

## 4.4 beemgraphenebase

### 4.4.1 beemgraphenebase package

#### Submodules

#### beemgraphenebase.account module

**class** beemgraphenebase.account.**Address**(*address=None*, *pubkey=None*, *prefix='STM'*)
Bases: object

Address class

This class serves as an address representation for Public Keys.

**Parameters**

- **address** (*str*) – Base58 encoded address (defaults to None)
- **pubkey** (*str*) – Base58 encoded pubkey (defaults to None)
- **prefix** (*str*) – Network prefix (defaults to STM)

Example:

```
Address("STMFN9r6VYzBK8EKtMewfNbfiGCr56pHDBFi")
```

**derive256address_with_version**(*version=56*)
Derive address using RIPEMD160(SHA256(x)) and adding version + checksum

**derivesha256address**()
Derive address using RIPEMD160(SHA256(x))

**derivesha512address**()
Derive address using RIPEMD160(SHA512(x))

**get_public_key**()
Returns the pubkey

**class** beemgraphenebase.account.**BrainKey**(*brainkey=None*, *sequence=0*)
Bases: object

Brainkey implementation similar to the graphene-ui web-wallet.

**Parameters**

- **brainkey** (*str*) – Brain Key
- **sequence** (*int*) – Sequence number for consecutive keys

Keys in Graphene are derived from a seed brain key which is a string of 16 words out of a predefined dictionary with 49744 words. It is a simple single-chain key derivation scheme that is not compatible with BIP44 but easy to use.

Given the brain key, a private key is derived as:

```
privkey = SHA256(SHA512(brainkey + " " + sequence))
```

Incrementing the sequence number yields a new key that can be regenerated given the brain key.

**get_blind_private**()
Derive private key from the brain key (and no sequence number)

**get_brainkey**()
Return brain key of this instance

**get_private**()
Derive private key from the brain key and the current sequence number

**get_private_key**()

**get_public**()

**get_public_key**()

**next_sequence**()
Increment the sequence number by 1

**normalize**(*brainkey*)
Correct formating with single whitespace syntax and no trailing space

**suggest**()
Suggest a new random brain key. Randomness is provided by the operating system using os.urandom().

**class** beemgraphenebase.account.**PasswordKey**(*account*, *password*, *role='active'*, *prefix='STM'*)
Bases: object

This class derives a private key given the account name, the role and a password. It leverages the technology of Brainkeys and allows people to have a secure private key by providing a passphrase only.

**get_private**()
Derive private key from the brain key and the current sequence number

**get_private_key**()

**get_public**()

**get_public_key**()

**class** beemgraphenebase.account.**PrivateKey**(*wif=None*, *prefix='STM'*)
Bases: *beemgraphenebase.account.PublicKey*

Derives the compressed and uncompressed public keys and constructs two instances of PublicKey:

**Parameters**

- **wif** (*str*) – Base58check-encoded wif key
- **prefix** (*str*) – Network prefix (defaults to STM)

Example::

```
PrivateKey("5HqUkGuo62BfcJU5vNhTXKJRXuUi9QSE6jp8C3uBJ2BVHtB8WSd")
```

Compressed vs. Uncompressed:

- **PrivateKey("w-i-f").pubkey:** Instance of PublicKey using compressed key.
- **PrivateKey("w-i-f").pubkey.address:** Instance of Address using compressed key.
- **PrivateKey("w-i-f").uncompressed:** Instance of PublicKey using uncompressed key.
- **PrivateKey("w-i-f").uncompressed.address:** Instance of Address using uncompressed key.

**child**(*offset256*)
Derive new private key from this key and a sha256 "offset"

**compressedpubkey**()
Derive uncompressed public key

**derive_from_seed**(*offset*)
Derive private key using "generate_from_seed" method. Here, the key itself serves as a *seed*, and *offset* is expected to be a sha256 digest.

**derive_private_key**(*sequence*)
Derive new private key from this private key and an arbitrary sequence number

**get_public_key**()
Returns the pubkey

**get_secret**()
Get sha256 digest of the wif key.

**class** beemgraphenebase.account.**PublicKey**(*pk*, *prefix='STM'*)
Bases: *beemgraphenebase.account.Address*

This class deals with Public Keys and inherits Address.

**Parameters**

- **pk** (*str*) – Base58 encoded public key
- **prefix** (*str*) – Network prefix (defaults to STM)

Example::

```
PublicKey("STM6UtYWWs3rkZGV8JA86qrgkG6tyFksgECefKE1MiH4HkLD8PFGL")
```

---

**Note:** By default, graphene-based networks deal with **compressed** public keys. If an **uncompressed** key is required, the method unCompressed can be used:

```
PublicKey("xxxxx").unCompressed()
```

**compressed**()
Derive compressed public key

**get_public_key**()
Returns the pubkey

**point**()
Return the point for the public key

**unCompressed**()
Derive uncompressed key

### beemgraphenebase.base58 module

**class** beemgraphenebase.base58.**Base58**(*data*, *prefix='GPH'*)
Bases: object

Base58 base class

This class serves as an abstraction layer to deal with base58 encoded strings and their corresponding hex and binary representation throughout the library.

**Parameters**

- **data** (*hex, wif, bip38 encrypted wif, base58 string*) – Data to initialize object, e.g. pubkey data, address data, ...
- **prefix** (*str*) – Prefix to use for Address/PubKey strings (defaults to GPH)

**Returns** Base58 object initialized with data

**Return type** *Base58*

**Raises** **ValueError** – if data cannot be decoded

- bytes(Base58): Returns the raw data
- str(Base58): Returns the readable Base58CheckEncoded data.
- repr(Base58): Gives the hex representation of the data.
- **format(Base58, _format) Formats the instance according to _format:**
  - "btc": prefixed with 0x80. Yields a valid btc address
  - "wif": prefixed with 0x00. Yields a valid wif key
  - "bts": prefixed with BTS
  - etc.

beemgraphenebase.base58.**b58decode**(*v*)

beemgraphenebase.base58.**b58encode**(*v*)

beemgraphenebase.base58.**base58CheckDecode**(*s*)

beemgraphenebase.base58.**base58CheckEncode**(*version*, *payload*)

beemgraphenebase.base58.**base58decode**(*base58_str*)

beemgraphenebase.base58.**base58encode**(*hexstring*)

beemgraphenebase.base58.**doublesha256**(*s*)

beemgraphenebase.base58.**gphBase58CheckDecode**(*s*)

beemgraphenebase.base58.**gphBase58CheckEncode**(*s*)

beemgraphenebase.base58.**log = <logging.Logger object>**
Default Prefix

beemgraphenebase.base58.**ripemd160**(*s*)

### beemgraphenebase.bip38 module

**exception** beemgraphenebase.bip38.**SaltException**
Bases: Exception

beemgraphenebase.bip38.**decrypt**(*encrypted_privkey*, *passphrase*)
BIP0038 non-ec-multiply decryption. Returns WIF privkey.

**Parameters**

- **encrypted_privkey** (Base58) – Private key
- **passphrase** (*str*) – UTF-8 encoded passphrase for decryption

**Returns** BIP0038 non-ec-multiply decrypted key

**Return type** *Base58*

**Raises** ***SaltException*** – if checksum verification failed (e.g. wrong password)

beemgraphenebase.bip38.**encrypt**(*privkey*, *passphrase*)
BIP0038 non-ec-multiply encryption. Returns BIP0038 encrypted privkey.

**Parameters**

- **privkey** (Base58) – Private key
- **passphrase** (*str*) – UTF-8 encoded passphrase for encryption

**Returns** BIP0038 non-ec-multiply encrypted wif key

**Return type** *Base58*

### beemgraphenebase.ecdasig module

### beemgraphenebase.objects module

**class** beemgraphenebase.objects.**GrapheneObject**(*data=None*)
Bases: object

Core abstraction class

This class is used for any JSON reflected object in Graphene.

- instance.__json__(): encodes data into json format
- bytes(instance): encodes data into wire format
- str(instances): dumps json object as string

**json**()

**toJson**()

**class** beemgraphenebase.objects.**Operation**(*op*)
Bases: object

**getOperationNameForId**(*i*)
Convert an operation id into the corresponding string

**operations**()

beemgraphenebase.objects.**isArgsThisClass**(*self*, *args*)

#### beemgraphenebase.objecttypes module

beemgraphenebase.objecttypes.**object_type = {'OBJECT_TYPE_COUNT': 3, 'account': 2, 'base':**
Object types for object ids

#### beemgraphenebase.operations module

beemgraphenebase.operationids.**operations = {'demooepration': 0}**
Operation ids

#### beemgraphenebase.transactions module

#### Module contents

beemgraphenebase.

## 4.5 beemgrapheneapi

### 4.5.1 beemgrapheneapi package

#### Submodules

#### GrapheneRPC

**Note:** This is a low level class that can be used in combination with GrapheneClient

This class allows to call API methods exposed by the witness node via websockets. It does **not** support notifications and is not run asynchronously.

**class** beemgrapheneapi.graphenerpc.**GrapheneRPC**(*urls*, *user=None*, *password=None*, ***kwargs*)
This class allows to call API methods synchronously, without callbacks.

It logs warnings and errors.

**Parameters**

- **urls** (*str*) – Either a single Websocket/Http URL, or a list of URLs
- **user** (*str*) – Username for Authentication

- **password** (*str*) – Password for Authentication
- **num_retries** (*int*) – Try x times to num_retries to a node on disconnect, -1 for indefinitely
- **num_retries_call** (*int*) – Repeat num_retries_call times a rpc call on node error (default is 5)
- **timeout** (*int*) – Timeout setting for https nodes (default is 60)

Available APIs

- database
- network_node
- network_broadcast

Usage:

---

**Note:** This class allows to call methods available via websocket. If you want to use the notification subsystem, please use `GrapheneWebsocket` instead.

---

**get_network**(*props=None*)
Identify the connected network. This call returns a dictionary with keys chain_id, core_symbol and prefix

**get_request_id**()
Get request id.

**get_use_appbase**()
Returns True if appbase ready and appbase calls are set

**is_appbase_ready**()
Check if node is appbase ready

**next**()
Switches to the next node url

**rpcclose**()
Close Websocket

**rpcconnect**(*next_url=True*)
Connect to next url in a loop.

**rpcexec**(*payload*)
Execute a call by sending the payload.

**Parameters** **payload** (*json*) – Payload data

**Raises**

- **ValueError** – if the server does not respond in proper JSON format
- **RPCError** – if the server returns an error

**rpclogin**(*user*, *password*)
Login into Websocket

### Module contents

beemgrapheneapi.

CHAPTER 5

# Glossary

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## b

# Index

## Symbols



## A



## B

## C

## D



## E



## F



## G

## H



## I



## J



## K



## L

## M



## N



## O



## P

## Q



## R



## S



## T

## U



## V



## W